

AD-A165 314

ADA (TRADEMARK) TRAINING CURRICULUM ADA (REGISTERED  
TRADEMARK) FOR SOFTWARE MANAGERS L201 TEACHER'S GUIDE  
VOLUME 1(U) SOFTECH INC WALTHAM MA 1986

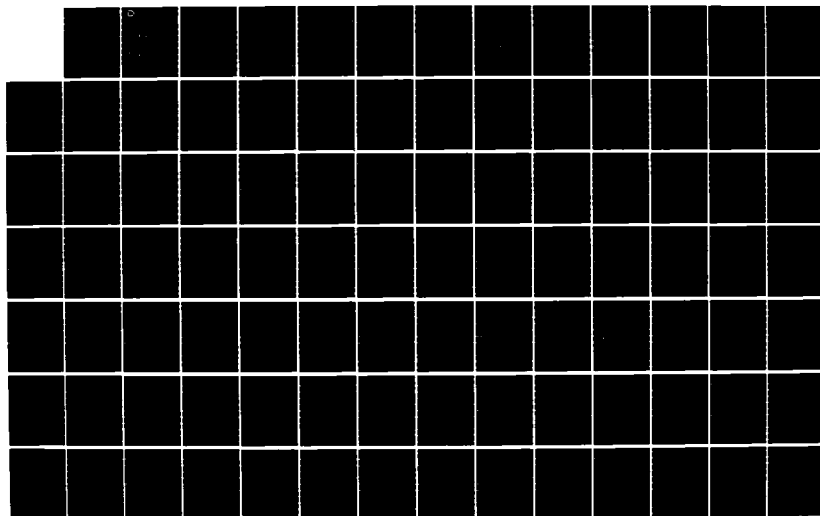
1/5

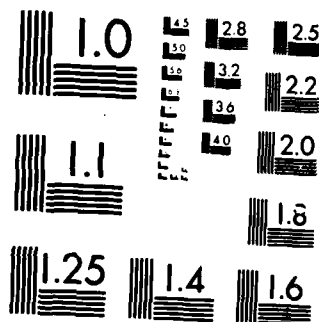
UNCLASSIFIED

DAAB07-83-C-K506

F/G 5/9

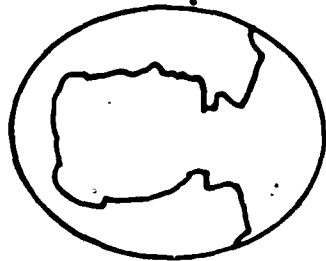
NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

DTIC FILE COPY



# Ada® Training Curriculum

AD-A165 314

## Ada® For Software Managers

L201

### Teacher's Guide Volume I

06 3 12 066

U.S. Army Communications-Electronics Command  
(CECOM)

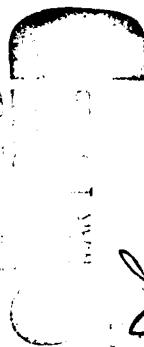
Contract DAAB07-83-C-K506

①

1986



DTIC



AD-A165 314

Prepared By:

SOFTECH, INC.  
460 Totten Pond Road  
Waltham, MA 02154

•Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

\*Approved For Public Release/Distribution Unlimited

## INSTRUCTOR NOTES

### GOALS:

- DEVELOP A CONCEPTUAL KNOWLEDGE OF ADA
- RECOGNIZE HIGH/POOR QUALITY DESIGNS AND CODE IN ADA
- DEVELOP AN UNDERSTANDING OF PORTABILITY AND REUSABILITY ISSUES

### THE FOLLOWING ARE NOT GOALS:

TO TEACH ADA DESIGN OR ADA CODING

### INTENDED AUDIENCE:

SOFTWARE MANAGERS  
DESIGNERS  
QA, CONTRACT MONITORS  
INTEGRATION

STRESS TO THE CLASS THAT THIS MODULE PRESUPPOSES AN INTRODUCTORY OVERVIEW OF THE ADA LANGUAGE IN THE NOT-100-DISTANT PAST. THAT IS, THE STUDENT HAD A RUDIMENTARY IDEA OF THE VARIOUS ADA FEATURES.

VG 823.1

i

Copyright by SofTech, Inc. 1984. This material may be reproduced by or for the U.S. Government pursuant to the copyright license under DAR clause 7-104.9(a)(May 81).



# **ADA FOR SOFTWARE MANAGERS**

## **L201**

VG 823.1

## INSTRUCTOR NOTES

COURSE OUTLINE FOR THE 3 DAYS. SECTION 1 IS A REFRESHER ON A BASIC ADA SYSTEM; SECTIONS 2-16 PRESENT EACH ADA FEATURE IN GREATER DETAIL WITH SHORT EXERCISES INTERSPERSED; SECTION 17 THEN USES THIS ADA KNOWLEDGE IN A LARGER EXERCISE AIMED AT ADA DESIGN AND CODE ASSESSMENT; SECTION 18 FORMALIZES THE DISCOVERIES OF SECTION 17; AND SECTION 19 SUMMARIZES THE ENTIRE COURSE BY PUTTING ADA INTO ITS PRIMARY DESIGN PURPOSE: MORE REUSABLE AND PORTABLE SOFTWARE.

# ADA FOR SOFTWARE MANAGERS

## Course Outline

INTRODUCTION  
(Section 1)

ADA FEATURES  
(Section 2-16)

INTRODUCTION TO ADA DESIGN/CODE ASSESSMENT  
(Section 17)

CHARACTERISTICS OF GOOD ADA DESIGNS  
(Section 18)

ADA IN PERSPECTIVE: REUSABILITY AND PORTABILITY  
(Section 19)

Accession For	
NTIS	GRA&I
DTIC	PC
A-1	



# INSTRUCTOR NOTES

## TIME MAP

Section 1: Introduction Section 2: Lexical Rules Section 3: Data Types
<b>BREAK</b>
Section 3: (Continued)
<b>LUNCH</b>
Section 4: Statements
<b>BREAK</b>
Section 5: Packages/Private Types

DAY 1 \_\_\_\_\_

## TIME MAP

Section 6: Subprograms
<b>BREAK</b>
Section 7: Tasks Section 8: Generics
<b>LUNCH</b>
Section 8: (continued) Section 9: I/O
<b>BREAK</b>
Section 10: Exceptions

DAY 2 \_\_\_\_\_

## TIME MAP

Section 11: Stubbing Section 12: Visibility and Scope Section 13: Overloading Section 14: Pragmas
<b>BREAK</b>
Section 16: Low-level Features Section 17: Introduction to Ada Design/Code Assessment (2½ hrs)
<b>LUNCH</b>
Section 17: (continued)
<b>BREAK</b>
Section 18: Characteristics of Good Ada Design Section 19: Ada in Perspective

DAY 3 \_\_\_\_\_

# **Section 1**

## **INTRODUCTION**

VG 823.1

## INSTRUCTOR NOTES

THIS SECTION QUICKLY REVIEWS THE BASICS OF AN ADA SYSTEM AND ITS COMPILATION. IF THIS IS TAUGHT IN CONJUNCTION WITH EITHER L101 OR L102 THIS SECTION COULD BE SKIPPED OR QUICKLY GLOSSED THROUGH.

# ADA FOR SOFTWARE MANAGERS

INTRODUCTION  
(Section 1)

ADA FEATURES  
(Section 2-16)

INTRODUCTION TO ADA DESIGN/CODE ASSESSMENT  
(Section 17)

CHARACTERISTICS OF GOOD ADA DESIGNS  
(Section 18)

ADA IN PERSPECTIVE: REUSABILITY AND PORTABILITY  
(Section 19)

## INSTRUCTOR NOTES

### POINT OUT:

1. TWO (2) PACKAGES ARE IMPORTED TO THE PROCEDURE (THE MAIN DRIVER HERE)
2. DECLARATIVE PART OF THE PROCEDURE AND WHAT IT SHOULD CONTAIN.  
(EXECUTABLE PART IS ON THE NEXT SLIDE.)
3. WITH VS. USE

BEFORE LOOKING AT INDIVIDUAL ADA FEATURES, WE QUICKLY REVIEW WHAT AN ADA SYSTEM AND  
COMPILATION LOOKS LIKE.



# AN ADA SYSTEM

## DECLARATIVE PART:

```
with Text_IO, Vector_Services;
use Vector_Services;
procedure Compute_Tracking_Data is

    Last_Point, Current_Point, Next_Point : Point_Type;
    Time_Elapsed, Time_Projected : Time_Type;
    Distance, Velocity : Float;

    package Time_IO is new Text_IO.Fixed_IO (Time_Type);
    package Flt_IO is new Text_IO.Float_IO (Float);

    procedure Get_Point (P : out Point_Type) is separate;
    procedure Put_Point (P : in Point_Type) is separate;

begin -- Compute_Tracking_Data
```

EXECUTABLE PART ON NEXT PAGE

```
end Compute_Tracking_Data;
```

INSTRUCTOR NOTES

THIS IS THE EXECUTABLE PART OF THE PROCEDURE WHOSE DECLARATIVE PART APPEARS ON THE PREVIOUS SLIDE. POINT OUT THAT THIS PART CONTAINS EXECUTABLE STATEMENTS (IN THIS CASE, PROCEDURE CALLS AND ASSIGNMENT STATEMENTS).

# AN ADA SYSTEM (Continued)

## EXECUTABLE PART:

procedure Compute\_Tracking\_Data is

### DECLARATIVE PART ON PREVIOUS PAGE

```
begin -- Compute Tracking_Data
  Text_IO.Put ("Enter coordinates of last position: ");
  Get_Point (Last_Point);
  Text_IO.Put ("Enter coordinates of current position: ");
  Get_Point (Current_Point);
  Text_IO.Put ("Time (in seconds) between readings : ");
  Time_IO.Get (Time_Elapsed); Text_IO.New_Line;
  Text_IO.Put ("Time (in seconds) until next reading : ");
  Time_IO.Get (Time_Projected); Text_IO.New_Line;
  Distance := Distance_Between (Last_Point, Current_Point);
  Calculate_Velocity (Last_Point, Current_Point, Time_Elapsed, Velocity);
  Next_Point := Next_Point_After (Last_Point, Current_Point,
    Time_Elapsed, Time_Projected);

  Text_IO.Put ("Distance between points was");
  Flt_IO.Put (Distance);
  Text_IO.Put_Line ("units.");
  Text_IO.Put ("Velocity was");
  Flt_IO.Put (Velocity);
  Text_IO.Put ("units per second.");
  Text_IO.Put ("After");
  Time_IO.Put (Time_Projected);
  Text_IO.Put ("seconds, the next point should be");
  Put_Point (Next_Point);

end Compute_Tracking_Data;
```

## INSTRUCTOR NOTES

### POINT OUT:

1. THIS IS THE PACKAGE SPECIFICATION FOR Vector\_Services USED BY OUR MAIN PROCEDURE.
2. PURPOSE OF A SPEC.
3. PACKAGE COLLECTS LOGICALLY RELATED PROCEDURES/FUNCTIONS (SUBPROGRAMS).
4. A SUBPROGRAM DECLARATION SPECIFIES ITS PURPOSE AND PARAMETERS FOR INTERFACE REQUIREMENTS WITH THE CALLER.

# AN ADA SYSTEM (Continued)

## PACKAGE SPECIFICATION:

```
package Vector_Services is
  type Coordinate_Type is (X,Y);
  type Point_Type is array (Coordinate_Type) of Float;
  subtype Time_Type is Duration;
  function Distance_Between (Last_Point, This_Point : Point_Type) return Float;
  procedure Calculate_Velocity (From, To : in Point_Type;
    In_Time : in Time_Type;
    Velocity : out Float);
  function Next_Point_After (Last_Point, This_Point : in Point_Type;
    Time_Between_Last, Time_Between_Next : Time_Type)
    return Point_Type;
end Vector_Services;
```

## INSTRUCTOR NOTES

### POINT OUT:

1. ALONG WITH A SPEC (THE INTERFACE) A package HAS AN ASSOCIATED BODY WHICH HAS THE IMPLEMENTATIONS OF THE SUBPROGRAMS OF THE SPEC.
2. 'is separate' SAYS WE'LL FIND THE ACTUAL CODE IN A SEPARATE PLACE (ALLOWS US TO SEE THE STRUCTURE OF THE BODY EASILY)
3. NOTICE THAT THE BODY CAN HAVE SUBPROGRAMS OTHER THAN THOSE SPECIFIED IN THE SPEC. Sqrt IS A UTILITY TO BE USED BY THE IMPLEMENTATION OF THE SUBPROGRAMS LISTED IN THE SPEC.

# AN ADA SYSTEM (Continued)

```
PACKAGE BODY:

package body Vector_Services is

function Sqrt (X : Float) return Float is separate;  -- BODY STUB

function Distance_Between (Last_Point, This_Point : Point_Type)
return Float is separate;

procedure Calculate_Velocity (From, To : in Point_Type;
    In_Time : in Time_Type;
    Velocity : out Float) is separate;

function Next_Point_After (Last_Point, This_Point : in Point_Type,
    Time_Between_Last, Time_Between_Next : Time_Type)
return Point_Type is separate;

end Vector_Services;
```

# INSTRUCTOR NOTES

HERE'S AN EXAMPLE OF AN ALGORITHM CODED IN ADA. IT IS A SQUARE ROOT CALCULATION. POINT  
OUT THAT ADA HAS loop AND if STATEMENTS SIMILAR TO OTHER LANGUAGES.



# AN ADA SYSTEM (Continued)

FUNCTION SUBUNIT (LOCAL TO PACKAGE BODY Vector\_Services):

```
separate (Vector_Services)
function Sqrt (X : Float) return Float is
  Epsilon : constant := 0.000001;
  Root    : Float := 1.0;
begin -- Sqrt
  if X = 0.0 then
    return 0.0;
  else
    Root := (X/Root + Root) / 2.0;
    while abs (X/Root**2 - 1.0) >= Epsilon
      loop
        Root := (X/Root + Root) / 2.0;
      end loop;
    return Root;
  end if;
end Sqrt;
```

## INSTRUCTOR NOTES

DON'T GO THROUGH THE CODE. THIS IMPLEMENTS THE CALCULATION OF THE LARGEST VALUE IN A LIST. POINT OUT:

1. IT'S A FUNCTION.
2. IT HAS AN EXPLICIT STATEMENT WHICH RETURNS A VALUE.
3. THE VALUE RETURNED MAY BE REPRESENTED BY AN EXPRESSION.

## AN ADA SYSTEM (Continued)

```
FUNCTION SUBUNIT (EXPORTED FROM PACKAGE Vector_Services):  
    separate (Vector_Services)  
    function Distance_Between (Last_Point, This_Point : Point_Type) return Float is  
        Dx, Dy : Float;  
    begin -- Distance_Between  
        Dx := abs (This_Point(X) - Last_Point(X));  
        Dy := abs (This_Point(Y) - Last_Point(Y));  
        return (Sqrt (Dx**2 + Dy**2) );  
    end Distance_Between;
```

## INSTRUCTOR NOTES

FOR REFERENCE AND COMPLETENESS ONLY, DON'T GO THROUGH CODE. (ALGORITHMS CALCULATE THE VELOCITY (USING THE DISTANCE FUNCTION) AND TO PREDICT THE NEXT POINT.)

# AN ADA SYSTEM (Continued)

SUBUNITS (EXPORTED FROM PACKAGE Vector\_Services):

separate (Vector\_Services)  
procedure Calculate\_Velocity (From, To : in Point\_Type;  
    In\_Time : in Time\_Type;  
    Velocity : out Float) is

begin -- Calculate Velocity  
    Velocity := Distance\_Between (From, To)/Float (In\_Time);  
end Calculate\_Velocity;

separate (Vector\_Services)  
function Next\_Point\_After (Last\_Point, This\_Point : in Point\_Type;  
    Time\_Between\_Last, Time\_Between\_Next : Time\_Type)  
    return Point\_Type is

    Next\_Point : Point\_Type;

begin -- Next\_Point\_After

    if Time\_Between\_Last = 0 then  
        return This\_Point;  
    else

        Next\_Point(X) := Last\_Point(X) + Float (Time\_Between\_Next/Time\_Between\_Last)  
            \* abs (This\_Point(X) - Last\_Point(X));

        Next\_Point(Y) := Last\_Point(Y) + Float (Time\_Between\_Next/Time\_Between\_Last)  
            \* abs (This\_Point(Y) - Last\_Point(Y));

        return Next\_Point;  
    end if;

end Next\_Point\_After;

## INSTRUCTOR NOTES

FOR REFERENCE AND COMPLETENESS ONLY, DON'T GO THROUGH CODE. (ROUTINES TO FORMAT INPUT AND OUTPUT OF POINT COORDINATES. POINT OUT THE USE OF THE in AND out MODE PARAMETERS.

# AN ADA SYSTEM (Continued)

PROCEDURE SUBUNITS (LOCAL TO MAIN PROCEDURE Compute\_Tracking\_Data):

```
    separate (Compute_Tracking_Data)
    procedure Get_Point (P : out Point_Type) is
    begin -- Get_Point
        Text_IO.Put (" X = ");
        Flt_IO.Get (P(X));
        Text_IO.Put (" Y = ");
        Flt_IO.Get (P(Y));
        Text_IO.New_Line;
    end Get_Point;
```

```
    separate (Compute_Tracking_Data)
    procedure Put_Point (P : in Point_Type) is
    begin -- Put_Point
        Text_IO.Put ("(");
        Flt_IO.Put (P(X));
        Text_IO.Put (",");
        Flt_IO.Put (P(Y));
        Text_IO.Put (")");
    end Put_Point;
```

INSTRUCTOR NOTES

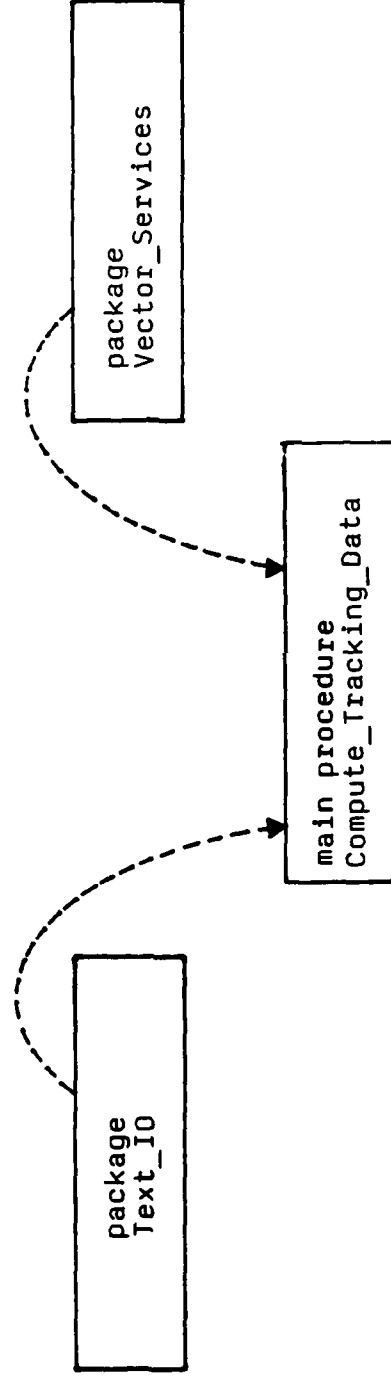
VG 823.1

1-10i



# SYSTEM COMPONENTS

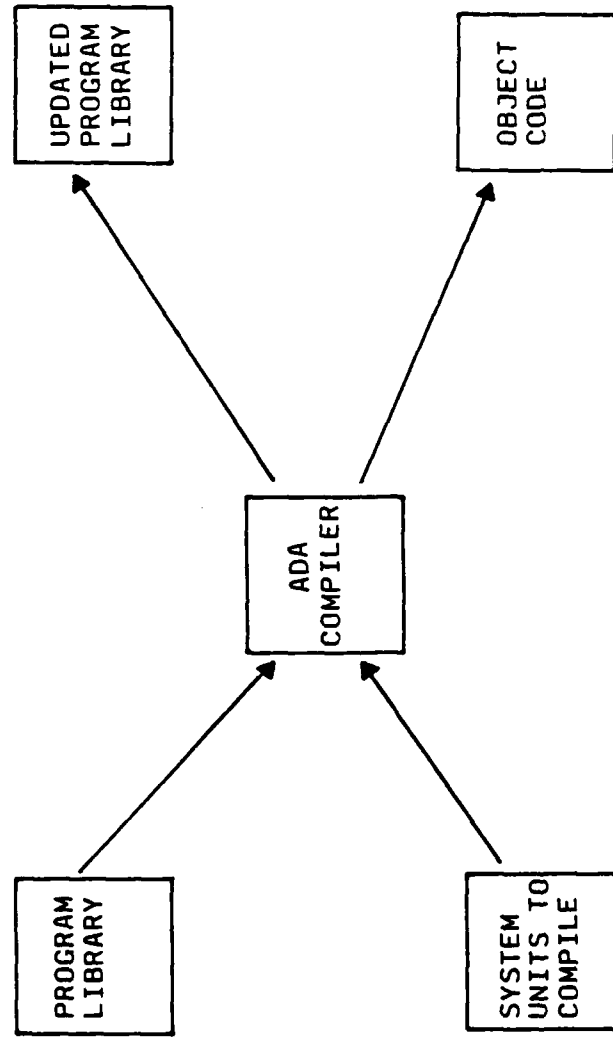
OUR EXAMPLE ADA SYSTEM IS COMPOSED OF:



INSTRUCTOR NOTES

THE PROGRAM LIBRARY IS A REPOSITORY OF PARTS OF AN ADA PROGRAM.

# TO COMPILE ANY ADA SYSTEM

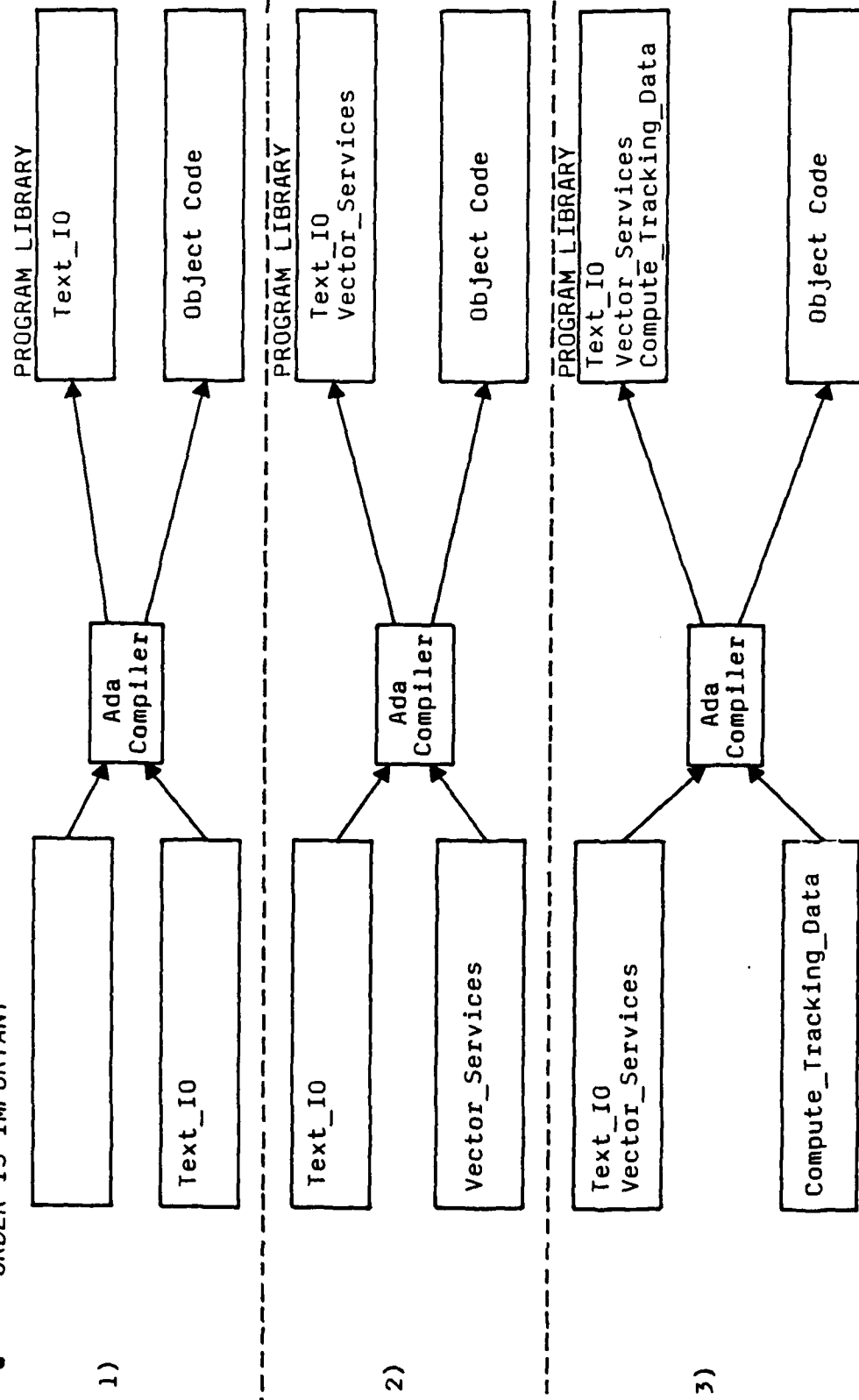


INSTRUCTOR NOTES

POINT OUT THAT STEPS 1 AND 2 COULD BE REVERSED BUT THEY BOTH MUST BE COMPILED INTO THE PROGRAM LIBRARY PRIOR TO STEP 3.

# TO COMPILE OUR EXAMPLE SYSTEM

- ORDER IS IMPORTANT



INSTRUCTOR NOTES

VG 823.1

1-131

# USE OF ADA FEATURES IN DESIGN

WITH POWERFUL FEATURES MUST COME RESPONSIBILITY. ADA FEATURES NEED TO BE USED  
WITH SPECIFIC INTENT AND KNOWLEDGE.

MANAGEMENT DECISIONS ON HOW ADA FEATURES WILL OR WILL NOT BE USED FOR YOUR  
PARTICULAR ORGANIZATION MAY BE NECESSARY.

INSTRUCTOR NOTES

VG 823.1

1-14i



# ADA FEATURES TO BE COVERED

<u>SECTION</u>	<u>FEATURE</u>
2	LEXICAL RULES
3	DATA TYPES
4	STATEMENTS
5	PACKAGES AND PRIVATE TYPES
6	SUBPROGRAMS
7	TASKS
8	GENERIC
9	INPUT/OUTPUT
10	EXCEPTIONS
11	STUBBING
12	VISIBILITY AND SCOPE
13	OVERLOADING
14	PRAGMAS
15	LOW-LEVEL FEATURES
16	SUMMARY OF USES OF ADA FEATURES

INSTRUCTOR NOTES

IN ADDITION TO LOOKING AT HOW TO USE THE FEATURES, THE "THINGS" TO WATCH OUT FOR ARE COVERED.

VG 823.1

1-151

# ADA FEATURES TO BE COVERED - FORMAT

SECTIONS 2-15 PRESENT A SURVEY OF POTENTIAL WAYS OR FORMS THAT PARTICULAR ADA FEATURES MIGHT BE USED. THIS IS NOT INTENDED TO BE A COMPLETE LIST OF USES.

EACH OF THESE SECTIONS IS FORMATTED AS FOLLOWS:

- THE NAME OF THE FEATURE
- A LIST OF USES OF THE FEATURE
- CONCEPTUAL RULES OF THE FEATURE
- AN EXAMPLE FOR EACH OF THE USES (GENERALLY)
- SOME POTENTIAL MISUSES OR PITFALLS (I.E., THINGS TO WATCH OUT FOR)

INSTRUCTOR NOTES

VG 823.1

# **ADA FEATURES (Sections 2-16)**

VG 823.1

INSTRUCTOR NOTES

VG 823.1

# ADA FOR SOFTWARE MANAGERS

INTRODUCTION  
(Section 1)

ADA FEATURES  
(Section 2-16)

INTRODUCTION TO ADA DESIGN/CODE ASSESSMENT  
(Section 17)

CHARACTERISTICS OF GOOD ADA DESIGNS  
(Section 18)

ADA IN PERSPECTIVE: REUSABILITY AND PORTABILITY  
(Section 19)

INSTRUCTOR NOTES

THIS SECTION, EUPHEMISTICALLY, DEALS WITH THE CARE AND FEEDING OF ADA FEATURES.

THIS SECTION IS THE LANGUAGE PORTION OF THE MODULE. WE WILL BE LOOKING AT ADA FEATURES AND CONSTRUCTS -- THEIR RULES, USES/MISUSES. WE WILL BE DEVELOPING A READING KNOWLEDGE OF ADA.



## **Section 2**

# **LEXICAL RULES**

VG 823.1

## INSTRUCTOR NOTES

LEXICAL ELEMENTS ARE THE BUILDING BLOCKS OF ADA STATEMENTS AND CLAUSES.

# LEXICAL ELEMENTS

- THE FUNDAMENTAL UNITS OF ADA CODE
- MUST BE USED TO WRITE LEGAL CODE
- CAN BE USED TO FORM MORE READABLE CODE

INSTRUCTOR NOTES

VG 823.1

2-21

# A PROGRAM IS A SEQUENCE OF LEXICAL ELEMENTS

- IDENTIFIERS
- NUMERIC LITERALS
- CHARACTER AND STRING LITERALS
- DELIMITERS
- COMMENTS

INSTRUCTOR NOTES

VG 823.1

2-31

# ADA CHARACTER SET

• THESE CHARACTERS ALL USED TO FORM LEXICAL ELEMENTS

## UPPER CASE

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

## LOWER CASE

a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z

## DIGITS

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

## SPECIAL CHARACTERS

"	#	%	&	'	(	)	*	+	,	-	.	/
:	;	<	=	>	_		!	\$	@	[	\	]
^	`	~	}	?								

## SPACE CHARACTER

INSTRUCTOR NOTES

THESE NAMES ARE FROM THE ADA SYSTEM IN SECTION 1.



# IDENTIFIERS

- USED AS NAMES

Vector\_Services  
Sqrt  
Calculate\_Velocity  
Current\_Point  
Index

-- A PACKAGE NAME  
-- A FUNCTION NAME  
-- A PROCEDURE NAME  
-- A VARIABLE NAME  
-- A LOOP PARAMETER NAME

# INSTRUCTOR NOTES

EXPLAIN THAT { } MEANS ZERO OR MORE REPETITIONS WHILE [ ] INDICATES AN OPTIONAL PART.

# RULES FOR IDENTIFIERS

## SYNTAX:

letter { [underscore] letter_or_digit }
---

NOTE: USE OF CURLY BRACKETS { } INDICATES OPTIONAL REPETITION.  
USE OF SQUARE BRACKETS [ ] INDICATES OPTIONAL PART.

- STARTS WITH A LETTER
- SUBSEQUENT CHARACTERS MAY BE LETTERS, DIGITS OR UNDERScores
- ALL CHARACTERS ARE SIGNIFICANT, INCLUDING UNDERScores
- NO EMBEDDED SPACES OR SPECIAL CHARACTERS
- MAY NOT CONTAIN CONSECUTIVE UNDERScores
- UPPER AND LOWER CASE ARE CONSIDERED THE SAME
- CAN BE AS LONG AS THE ENTIRE LINE

INSTRUCTOR NOTES

VG 823.1

2-6i

# EXAMPLES OF IDENTIFIERS

Calculate_Velocity	-- UNDERSCORE IS SIGNIFICANT
CalculateVelocity	-- NOT THE SAME AS Calculate_Velocity
Sqrt	
NUMBER_OF_ITEMS	-- NO DISTINCTION MADE BETWEEN
Number_of_Items	-- UPPER AND LOWER CASE
Size_30	-- IDENTIFIER MAY INCLUDE DIGITS
Extended_Security_Classification_Variant_Record_Type	-- A VERY LONG IDENTIFIER

INSTRUCTOR NOTES

VG 823.1

2-71

# IDENTIFIERS

- INCLUDE RESERVED WORDS
- RESERVED WORDS MAY BE WRITTEN IN EITHER LOWER CASE OR UPPER CASE  
(CONVENTION IS TO WRITE IN LOWER CASE)
- RESERVED WORDS CANNOT APPEAR AS USER-DEFINED IDENTIFIERS

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	task
all	do	in		terminate
and		is	package	then
array	else		pragma	type
at	elsif		private	
	end	limited	procedure	
begin	entry	loop		use
body	exception		raise	when
	exit	mod	range	while
			record	with
			rem	
case		new	renames	
constant	for	not	return	xor
	function	null	reverse	

# INSTRUCTOR NOTES

## POINT OUT:

1. IN NUMERIC LITERALS THE UNDERSCORE IS NOT SIGNIFICANT AND SO DOES NOT CREATE A NEW DISTINCT LITERAL. THIS IS DIFFERENT THAN THE RULES FOR IDENTIFIERS.
2. AN UNDERSCORE IS USED IN NUMERIC LITERALS FOR READABILITY.
3. A DIGIT MUST PRECEDE THE DECIMAL POINT IN REAL LITERALS.



# NUMERIC LITERALS

- REPRESENT NUMERIC VALUES

- INTEGER LITERALS

- NO DECIMAL POINT
  - OPTIONAL EXPONENT MUST BE ZERO OR POSITIVE INTEGER
  - EXAMPLES:

- 2500            -- ONE WAY TO WRITE 2,500
    - 2\_500        -- ALTERNATIVE WAY TO WRITE 2,500
    - 25E2        -- ANOTHER ALTERNATIVE WAY TO WRITE 2,500

- REAL LITERALS

- HAVE A DECIMAL POINT (CAN'T BE FIRST OR LAST)
  - OPTIONAL EXPONENT CAN BE ANY INTEGER
  - EXAMPLES:

- 12.75            -- ONE WAY TO WRITE 12.75
    - 0.1275E2        -- ALTERNATIVE WAY TO WRITE 12.75
    - 1\_275.0E-2      -- ANOTHER ALTERNATIVE WAY TO WRITE 12.75

# INSTRUCTOR NOTES

## POINT OUT:

1. TO OBTAIN THE CHARACTER LITERAL APOSTROPHE ('), SIMPLY ENCLOSE IT IN SINGLE APOSTROPHES (''').).
2. TO OBTAIN THE QUOTE (") WITHIN A STRING LITERAL, IT MUST BE DOUBLED (""").).

# CHARACTER LITERALS AND STRING LITERALS

- CHARACTER LITERALS

- FORMED BY ENCLOSING ONE CHARACTER BETWEEN SINGLE APOSTROPHE

## CHARACTERS

- ' ' AND ''' ARE VALID CHARACTER LITERALS

- EXAMPLE:

```
with Text_IO;  
procedure Output_Prompt is  
begin -- Output_Prompt  
  Text_IO.Put(' '); -- WRITE THE PROMPT  
end Output_Prompt;
```

- STRING LITERALS

- SEQUENCE OF ZERO OR MORE CHARACTERS ENCLOSED IN QUOTES

- EXAMPLE:

```
with Text_IO;  
procedure Welcoming_Message is  
begin -- Welcoming_Message  
  Text_IO.Put ("Welcome to Ada");  
  Text_IO.Put ("We claim, "It's not that tough!");  
end Welcoming_Message;
```

INSTRUCTOR NOTES

2-101

VG 823.1

# DELIMITERS

## SPECIAL CHARACTERS

& " ( ) \* + , . ' - |  
/ : ; < = > # -

## COMPOUND SYMBOLS

:=	ASSIGNMENT
..	RANGE DEFINITION
**	EXPONENTIATION OPERATION
>= <= /=	RELATIONAL OPERATORS
<< >>	IDENTIFIES STATEMENT LABELS
=>	INDICATES RELATIONSHIP BETWEEN A NAME AND A VALUE,
	ACTION, OR DECLARATION
<>	STANDS FOR INFORMATION TO BE FILLED IN LATER
--	COMMENT

## SPACE CHARACTER

INSTRUCTOR NOTES

VG 823.1

2-11i

# COMMENTS

- START WITH TWO HYPHENS AND ARE TERMINATED BY THE END OF THE LINE
- MAY NOT PRECEDE OTHER LEXICAL UNITS ON THE SAME LINE
- HAVE NO EFFECT ON THE MEANING OF THE PROGRAM.
- PURPOSE IS TO 'ENLIGHTEN' THE READER.

## EXAMPLE:

```
-- Exchange First_Value and Second_Value only if
-- First_Value is greater than Second_Value
if First_Value > Second_Value then
    Temp      := First_Value;
    First_Value := Second_Value;
    Second_Value := Temp;
end if;
-- First_Value <= Second_Value
```

# INSTRUCTOR NOTES

ALLOW FIVE MINUTES FOR THIS EXERCISE. ANSWERS TO THE QUESTIONS APPEAR BELOW.

1. Too\_Bad\_This\_Is\_Not\_An\_Identifier an identifier cannot start with a number
  2. Security\_Classification\_Type missing underscore
  3. OK
  4. Reserved Word
  5. Char\_Count
  6. OK
  7. -- of an incoming message
  8. 3.14 -- An ...
  9. 3.14
  10. 300\_000
  11. while not
  12. X := 4;
  13. "CAR NAME MILES PER GALLON"  
or  
""CAR NAME"" MILES PER GALLON" illegal quoted string in string
  14. 300E2 or 300.OE-2 Integer cannot be raised to negative power
- illegal double underscore
- need -- on second line
- need -- before comment
- underscore must be followed by a number
- illegal, use \_ for readability, not,
- reserved word, needs a space
- illegal space between : and =. illegal : at end



# CLASS EXERCISE

INDICATE WHAT, IF ANYTHING, IS WRONG WITH THE FOLLOWING PROGRAM ELEMENTS, AND  
CORRECT THOSE IN ERROR.

1. 2\_Bad\_This\_Is\_Not\_An\_Identifier
2. Security Classification\_Type
3. Channel\_Mode
4. loop
5. CHAR\_COUNT
6. x
7. -- The purpose of this procedure is to validate the security prosign  
of an incoming message
8. 3.14 - An abbreviated definition of PI
9. 3\_.14
10. 300,000 -- speed of light (km/sec)
11. whilenot
12. X : = 4:
13. ""CAR NAME" MILES PER GALLON"
14. 300E-2

INSTRUCTOR NOTES

VG 823.1

2-131

# LEXICAL STYLE

- FREE FORMAT LANGUAGE - USE INDENTATION AND BLANK LINES FOR READABILITY.
- LEXICAL ELEMENTS MUST FIT ON ONE LINE.
- SPACES OPTIONAL BETWEEN MOST LEXICAL ELEMENTS BUT MANDATORY BETWEEN TWO LEXICAL ELEMENTS WHICH WITHOUT SEPARATING SPACE COULD BE CONSTRUED AS ONE LEXICAL ELEMENT.
- NO CONTINUATION MARKS (I.E. STATEMENTS MAY CROSS LINE BOUNDARIES).
- LINE MAY CONTAIN MORE THAN ONE STATEMENT.
- STATEMENT MUST BE TERMINATED BY A SEMICOLON.

# INSTRUCTOR NOTES

HAVE STUDENTS IDENTIFY THE VARIOUS LEXICAL ELEMENTS THAT WE HAVE SEEN SO FAR. ALSO HAVE THEM IDENTIFY ANYTHING ILLEGAL.

HINT: THIS INCLUDES FREE FORMAT.

ALLOW 5 MINUTES FOR THIS.

```

procedure Calculate_Median
(List: in List_Type;
 Midpoint: out Scores_Type) is
    Index_1,
    Index_2: Positive;
    Temp_List : List_Type := List;
begin -- Calculate_Median
    Index_1 := (Temp_List'Last+1)/2;
    Index_2 := (Temp_List'Last/2)
    + 1;
    Sort(Temp_List); Midpoint:= (Temp_List(Index_1) + Temp_List(Index_2))/2.0;
end Calculate_
Median;

```

blank lines

indentation

statement

crossing line

boundaries

multiple statements per line

end Calculate\_ Median;

\*\* illegal, identifiers may not cross line boundaries

(THIS ALGORITHM CALCULATES THE MEDIAN OR PHYSICAL MIDPOINT OF A SORTED LIST OF VALUES).

NOTE: List\_Type IS DEFINED AS AN ARRAY OF REAL NUMBERS.

# CLASS EXERCISE

```
procedure Calculate_Median
(List: in List_Type;
 Midpoint: out Scores_Type) is
    Index_1,
    Index_2: Positive;
    Temp_List : List_Type := List;
begin -- Calculate_Median

    Index_1 := (Temp_List'Last+1)/2;
    Index_2 := (Temp_List'Last/2)
               +1;

    Sort(Temp_List); Midpoint:= (Temp_List(Index_1) + Temp_List(Index_2))/2.0;

end Calculate_
Median;
```

# INSTRUCTOR NOTES

POINT OUT THE ALIGNMENT OF VARIABLES, THE VERTICAL SPACING, AND THE CHOICE OF IDENTIFIER NAMES TO REFLECT THEIR FUNCTION.

AGAIN, A LIST IN THIS CONTEXT IS AN ARRAY OF REAL NUMBERS.

# LEXICAL ELEMENTS TO INCREASE READABILITY AND MAINTAINABILITY

procedure Sort (List : in List\_Type) is

Temp : Scores\_Type;  
Sorted : Boolean := False;

begin -- Sort

PHYSICAL NESTING MIRRORS  
LOGICAL NESTING

while not Sorted loop

Sorted := True;

for I in List'First .. List'Last - 1 loop

if List (I) > List (I + 1) then

-- EXCHANGE NEXT ELEMENT FOR CURRENT ELEMENT

Temp := List (I + 1);

List (I + 1) := List (I);

List (I) := Temp;

Sorted := False;

end if;

end loop; -- FOR

end loop; -- WHILE

end Sort;

COMMENTS TO AUGMENT CODE

INSTRUCTOR NOTES

VG 823.1

2-161



# SOME POTENTIAL PITFALLS - LEXICAL

- LACK OF DISTINCT AND MEANINGFUL NAMES FOR ALL THE ENTITIES TO BE NAMED IN AN ADA SYSTEM (TYPES, PROGRAM UNITS, VARIABLES, CONSTANTS)

- SHORT, CRYPTIC IDENTIFIERS

Q\_Head ← Queue\_Head  
GENCC ← Generated\_Control\_Character\_Task

- TOO MANY COMMENTS, RATHER THAN CAREFUL CHOICE OF ENTITY NAMES

-- IDENTIFIERS CAN BE SELF-DOCUMENTING FOR THE PROGRAM READER

Begin\_Alt ← Start\_Using\_Alternates  
Is\_Kill ← Kill\_Flag\_Is\_Set

- LITTLE OR INAPPROPRIATE USE OF FREE FORMATTING TO INCREASE THE READABILITY OR LOGICAL UNDERSTANDING OF A PIECE OF CODE

INSTRUCTOR NOTES

VG 823.1

3-i

# **Section 3**

## **DATA TYPES**

VG 823.1

## INSTRUCTOR NOTES

THE FIRST SEVENTY PAGES OR SO (ABOUT THREE (3) HOURS WORTH) DISCUSS THE DIFFERENT CLASSES OF TYPES AND THEIR ASSOCIATED OPERATIONS, INCLUDING EXAMPLES AND SCATTERED EXERCISES. THE PITFALLS FOLLOW THIS OVERVIEW OF TYPES.

# ADA TYPE SYSTEM

CAN BE USED TO

- CORRECT LARGE CLASS OF ERRORS BEFORE TESTING/INTEGRATION
- DOCUMENT THE DESIGN
- EXPRESS DESIGN CONSTRAINTS OF OBJECTS

# INSTRUCTOR NOTES

AN ADA OBJECT IS WHAT IN MOST LANGUAGES IS CALLED A VARIABLE. WE WILL SEE IN A MOMENT WHY ADA USES THE NEUTRAL TERM "OBJECT."

# OBJECTS

- OBJECTS ARE CONTAINERS THAT HOLD DATA. TWO KINDS OF OBJECTS IN ADA:
  - VARIABLES: CAN BE ASSIGNED VALUES DURING THE PROGRAM
  - CONSTANTS: INITIAL VALUE ASSIGNED AT THE BEGINNING OF A PROGRAM UNIT WHICH CANNOT BE CHANGED
- IN ADDITION TO A VALUE, EVERY OBJECT HAS A PROPERTY CALLED TYPE, WHICH DESCRIBES THE KIND OF DATA THAT THE OBJECT MAY HOLD AND THE ALLOWED OPERATIONS.

INSTRUCTOR NOTES

THIS ILLUSTRATES SOME OF THE CONCEPTS OF TYPE. 3.5 CAN BE ASSIGNED TO Average\_Score; 2  
CAN'T BE SINCE IT IS NOT ONE OF THE ALLOWED VALUES.



# OBJECTS

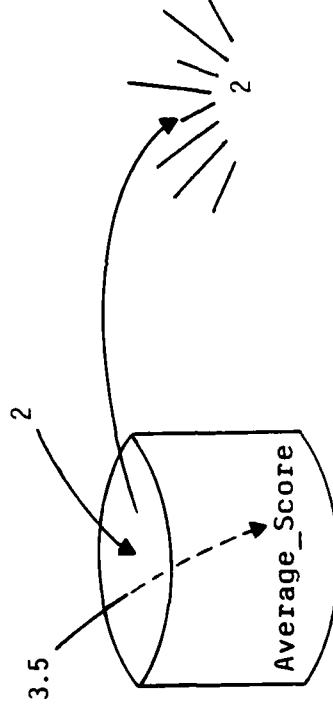
IF WE HAVE THE OBJECT DECLARATION

```
Average_Score : Float;
```

THEN BY DEFINITION OF TYPE Float, Average\_Score MUST HAVE DECIMAL POINT

Average\_Score IS AN OBJECT OF TYPE Float

THEN



INSTRUCTOR NOTES

VG 823.1

3-41

# OBJECTS

- ALL OBJECTS MUST BE DECLARED IN ADA
- OBJECTS ARE DECLARED IN THE DECLARATIVE PART

<pre>with Text_IO; use Text_IO; procedure Echo is   Char : Character;   .   . begin -- Echo   Get (Char);   Put (Char); end Echo;</pre>	<pre>} } }</pre>	<p>DECLARATIVE PART</p> <p>EXECUTABLE PART</p>
---	------------------	--

## INSTRUCTOR NOTES

BY IMPOSING THIS STRUCTURE ON OBJECTS WE CAN IMPROVE THE MAINTAINABILITY (DESCRIBE OBJECTS), READABILITY (DESCRIBE PROPERTIES), RELIABILITY (PROPERTY ADHERENCE), AND REDUCE COMPLEXITY (HIDE IMPLEMENTATION).

# TYPES

- REPRESENT THE FORM AND BEHAVIOR OF REAL-WORLD STRUCTURES IN THE PROGRAMMING LANGUAGE
- IMPOSE STRUCTURE ON DATA OBJECTS BY
  - DESCRIBING OBJECTS
  - DESCRIBING THEIR PROPERTIES
  - GUARANTEEING THAT THESE PROPERTIES ARE ADHERED TO
  - HIDING IMPLEMENTATION DETAILS

INSTRUCTOR NOTES

VG 823.1

3-61

# TYPES

- SOME TYPES ARE GIVEN TO US AS PART OF THE LANGUAGE
- SOME TYPES WILL BE USER-DEFINED
- A TYPE IS CHARACTERIZED BY:
  - A SET OF VALUES
  - A SET OF OPERATIONS APPLICABLE TO THOSE VALUES

## INSTRUCTOR NOTES

THE DIAGRAM PROVIDES A SUMMARY OF THE ADA TYPING STRUCTURE.

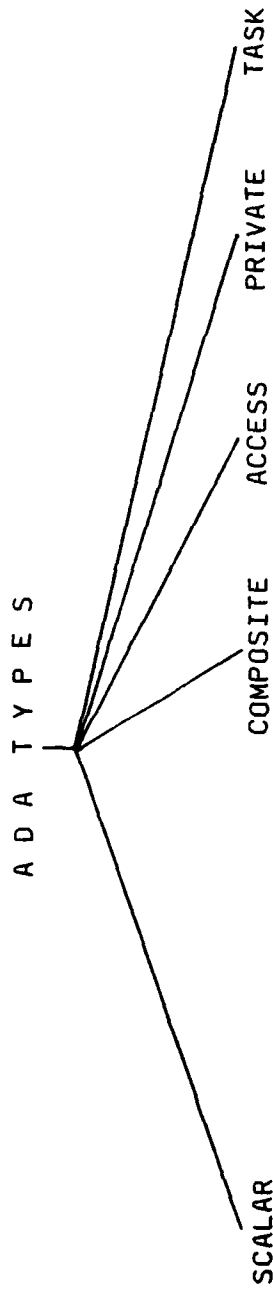
### EXAMPLES:

- SCALARS - INTEGERS, REALS
- COMPOSITES - ARRAYS
- ACCESS - POINTERS

WE'LL LOOK AT PRIVATE AND TASK TYPES IN LATER SECTIONS.



# CLASSES OF ADA TYPES

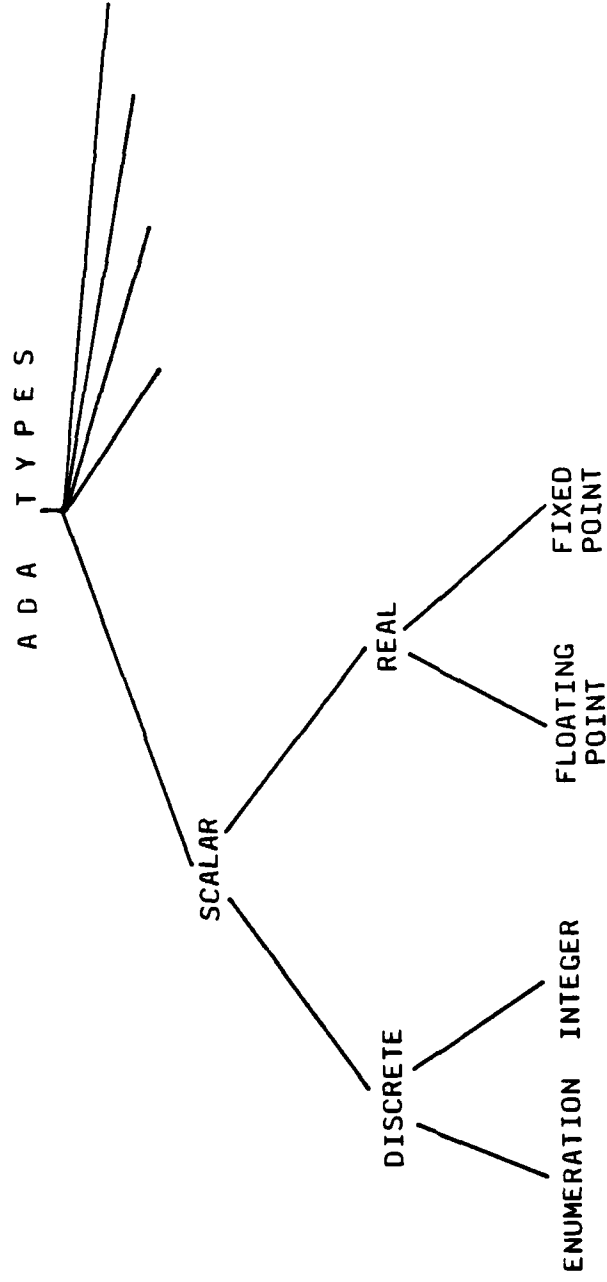


- SCALAR VALUES HAVE NO COMPONENTS
- COMPOSITE VALUES ARE MADE UP OF COMPONENTS
- ACCESS VALUES GIVE ACCESS TO OTHER OBJECTS
- PRIVATE TYPES ARE USED TO PROTECT HOW A TYPE IS IMPLEMENTED

## INSTRUCTOR NOTES

- A SCALAR IS AN ENTITY WHICH CANNOT BE DECOMPOSED ANY FURTHER.
- OBJECTS CAN ONLY ASSUME ONE VALUE AT A TIME
- SCALAR TYPE MAY BE EITHER "DISCRETE" OR "REAL"
  - DISCRETE SCALAR TYPES
    - Enumeration (e.g., Red, Blue Green)
    - Integer
  - REAL SCALAR TYPES
    - Floating
    - Fixed
- ALL SCALAR TYPES ARE ORDERED
  - EVERY VALUE IN THE TYPE IS EITHER LESS THAN OR GREATER THAN EVERY OTHER VALUE
  - DISCRETE VALUES HAVE IMMEDIATE PREDECESSORS AND SUCCESSORS

# SCALAR TYPES



- DISCRETE VALUES HAVE IMMEDIATE SUCCESSORS OR PREDECESSORS.

## INSTRUCTOR NOTES

- INTEGER TYPES

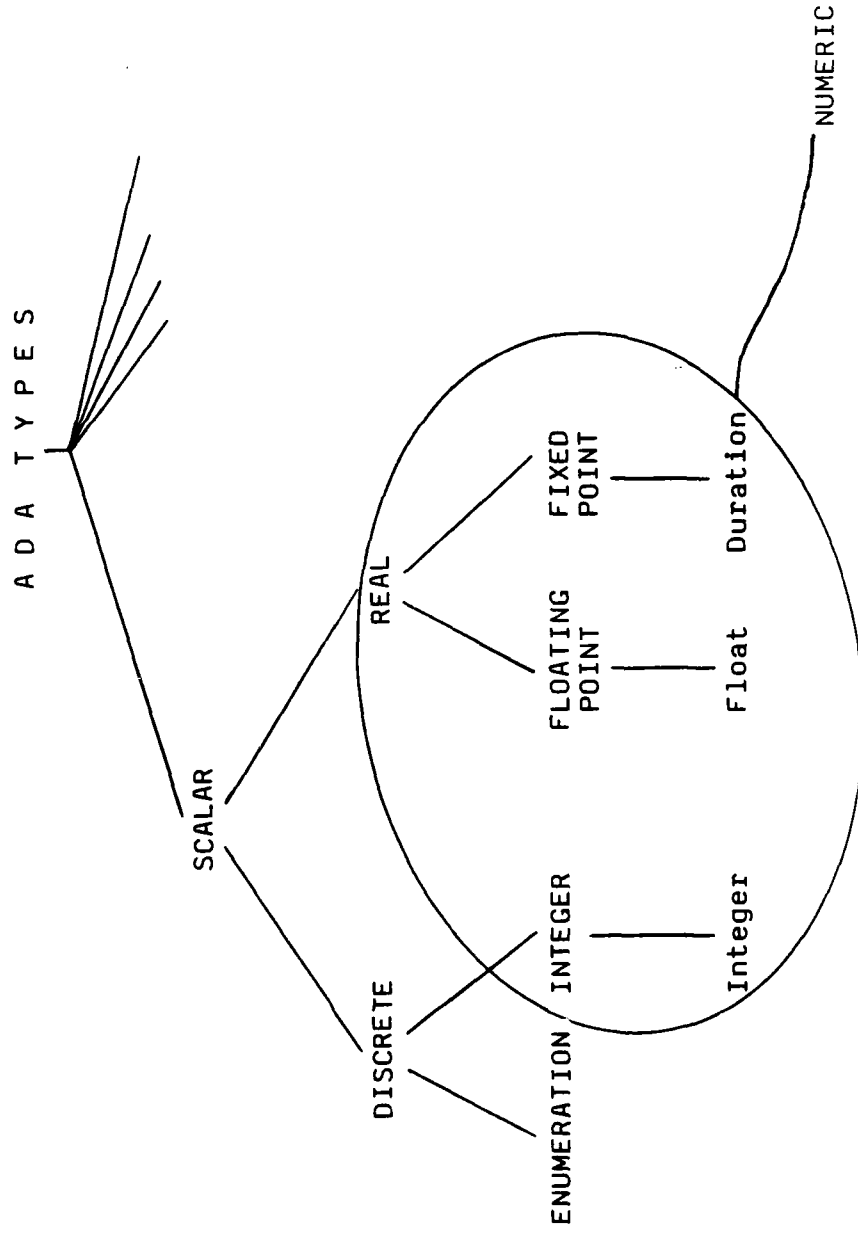
- INCLUDE THE PREDEFINED TYPE Integer WHOSE SET OF VALUES IS DEFINED BY A PARTICULAR HARDWARE IMPLEMENTATION
- HAVE NO DECIMAL POINT
- HAVE POSITIVE INTEGER EXPONENT ONLY

- REAL TYPES

- ARE FLOATING POINT TYPES OR FIXED POINT TYPES
- INCLUDE THE PREDEFINED TYPE Float WHOSE SET OF VALUES IS DEFINED BY A PARTICULAR HARDWARE IMPLEMENTATION
- MUST HAVE DECIMAL POINT
- HAVE POSITIVE OR NEGATIVE INTEGER EXPONENTS
- IN CASE SOMEONE ASKS, Duration IS A PREDEFINED FIXED POINT TYPE USED TO REPRESENT TIME

# NUMERIC TYPES

INTEGER AND REAL SCALAR TYPES ARE COLLECTIVELY KNOWN AS NUMERIC TYPES.



## INSTRUCTOR NOTES

- range Lower\_Bound .. Upper\_Bound IS CALLED RANGE CONSTRAINT (I.E. THE RANGE OF VALUES OBJECTS OF THE TYPE CAN HAVE)
- WHY USE USER-DEFINED TYPES? FOR MY PARTICULAR HARDWARE, PAGE NUMBER CAN'T BE GREATER THAN 2000 OR LESS THAN 1 SO I CAN REPRESENT THIS VIEW OF THE WORLD IN ADA. IF I DON'T NEED THE FULL RANGE OF INTEGER VALUES I DON'T NEED A FULL REPRESENTATION. IN THIS WAY THE IMPLEMENTATION IS FREE TO CHOOSE THE MOST EFFICIENT REPRESENTATION FOR THE TYPE.
- RANGE CONSTRAINT, IS MANDATORY FOR USER-DEFINED INTEGER TYPES.
- POINT OUT THAT A TYPE DECLARATION JUST DESCRIBES A TEMPLATE, TO ACTUALLY HAVE AN OBJECT WE MUST CREATE IT IN AN OBJECT DECLARATION. INDICATE THE SYNTAX FOR THIS IN THIS EXAMPLE. (NOTE USE OF THIS THROUGHOUT THE TYPE SECTION).

AD-A165 314

ADA (TRADEMARK) TRAINING CURRICULUM ADA (REGISTERED  
TRADEMARK) FOR SOFTWARE MANAGERS L201 TEACHER'S GUIDE  
VOLUME 1(U) SOFTECH INC WALTHAM MA 1986

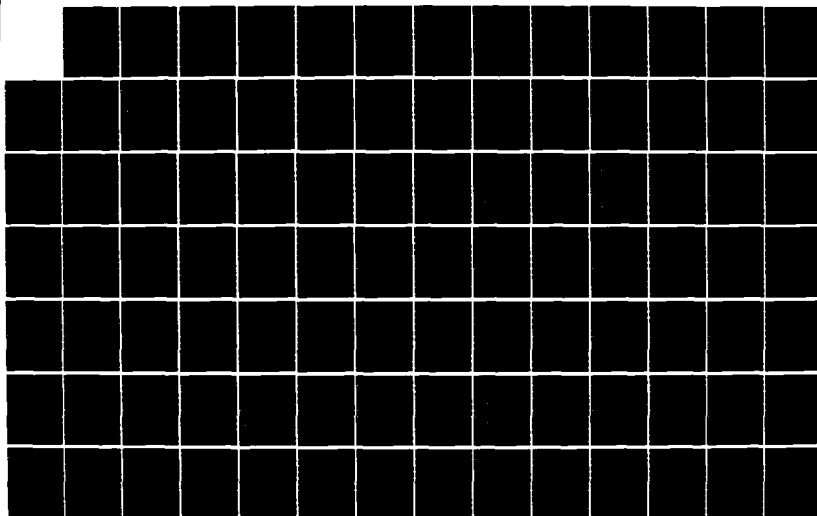
2/5

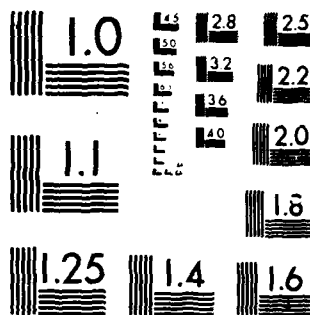
UNCLASSIFIED

DADB07-83-C-K506

F/G 5/9

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A



# INTEGER TYPES

- THE PREDEFINED TYPE Integer BELONGS TO THE CLASS OF INTEGER TYPES  
EXAMPLES OF OBJECTS OF TYPE INTEGER:

```
Count : Integer;  
Cruising_Altitude : Integer range 15_000 .. 35_000;  
Days_In_April : constant Integer := 30;
```

- USER CAN DEFINE INTEGER TYPES

SYNTAX:

```
type Identifier is range Lower_Bound .. Upper_Bound;
```

EXAMPLES:

```
type Page_Number_Type is range 1 .. 2000;  
Page_Number : Page_Number_Type;  
End_Page_Number : Page_Number_Type := 10;
```

THE IMPLEMENTATION CHOOSES THE REPRESENTATION OF THE TYPE BASED ON THE RANGE; THUS, PORTABILITY IS NOT AFFECTED

INSTRUCTOR NOTES

RELATE THE REAL-WORLD VIEW OF THE CODE.

NOTE RANGE CONSTRAINT IS OPTIONAL.

# REAL TYPES - FLOATING POINT

- RELATIVE BOUNDS CAN BE SPECIFIED

## SYNTAX:

type Identifier is digits N [range Lower\_Bound .. Upper\_Bound];

NOTE: N REPRESENTS THE MINIMUM NUMBER OF SIGNIFICANT DECIMAL  
DIGITS THAT MUST BE SUPPORTED BY THE HARDWARE

## EXAMPLES:

type Scores is digits 5 range 0.0 .. 1500.0;  
type Sensor\_Reading\_Type is digits 5;  
Score : Scores;  
Velocity, Altitude, Acceleration : Float;  
Temperature : Float := 273.0;

INSTRUCTOR NOTES

CONCEPTUALLY FIXED POINTS ARE DISCRETE REAL TYPES.

NOTE RANGE CONSTRAINT IS MANDATORY HERE.

Voltage\_Type DEFINES VALUES FROM -12.0 TO 12.0 IN STEPS OF 0.125.

# REAL TYPES - FIXED POINT

- ABSOLUTE ERROR BOUNDS CAN BE SPECIFIED

SYNTAX:

type Identifier is delta D range Lower\_Bound .. Upper\_Bound;

NOTE: D REPRESENTS THE DEGREE OF ACCURACY AND MUST BE A POSITIVE REAL

NUMBER

EXAMPLES:

type Voltage\_Type is delta 0.125 range -12.0 .. 12.0;

Interval: constant := 0.01;

type Fraction\_Type is delta Interval range 0.0 .. 1.0 - Interval;

Battery\_Voltage : Voltage\_Type := 11.875;

INSTRUCTOR NOTES

IN GENERAL, IT IS BETTER PROGRAMMING PRACTICE TO USE A NAMED NUMBER RATHER THAN A  
NUMERIC CONSTANT.

# NAMED NUMBERS

- SYMBOLIC NAMES FOR NUMBER VALUES (ACTS LIKE A NAMED LITERAL)
- INCREASES READABILITY AND EASE OF MAINTENANCE BY REFERENCING THE NAME THAT REFLECTS A LITERALS PURPOSE
- DECLARED IN NUMBER DECLARATIONS:

SYNTAX:

```
Name { , Name } : constant := Numeric_Value;
```

EXAMPLE:

```
Pi : constant := 3.14;
```

- DIFFERENCES BETWEEN NAMED NUMBER AND CONSTANTS:
  - CONSTANTS HAVE A TYPE SPECIFIED
  - NAMED NUMBERS TAKE THE TYPE FROM THE LITERAL

INSTRUCTOR NOTES

FOR EXAMPLE: IT DOESN'T MAKE SENSE TO ADD Page\_Number to Cruising\_Altitude.



# TYPE EQUIVALENCE

VALUES OF ONE TYPE CANNOT BE ASSIGNED TO VARIABLES OF A DIFFERENT TYPE

- PROGRAM REFLECTS ABSTRACT LOGICAL ENTITIES OF THE PROBLEM
- COMPILER AND RUN-TIME SYSTEM DETECTS ERRORS CAUSED BY TYPE MISMATCH (BETWEEN VARIABLE AND VALUE)
- COMPILER AND RUN-TIME SYSTEM CHECKS FOR SOFTWARE CONSISTENCY
- FOR EXAMPLE, NORMALLY IT IS ILLEGAL TO ADD AN Integer VALUE TO A Float VALUE

INSTRUCTOR NOTES

IN THE REAL WORLD CONVERSIONS ARE NECESSARY. ADA ALLOWS THIS FOR NUMERIC TYPES IN A SPECIFIC WAY.

# NUMERIC TYPE CONVERSIONS

## SYNTAX:

type\_name (expression)

## CONTEXT:

type Civil\_Aviation\_Frequencies is delta 0.025 range 108.0 .. 135.950;

I : Integer := 120;

Tower\_Frequency : Civil\_Aviation\_Frequencies;

## EXAMPLES:

Tower\_Frequency := Civil\_Aviation\_Frequencies (I);

INSTRUCTOR NOTES

BESIDES A SET OF VALUES AND OPERATION, ADDITIONAL INFORMATION ABOUT A TYPE IS AVAILABLE.

WE'LL LOOK AT SOME EXAMPLES NEXT.

# ATTRIBUTES

ADA OFFERS NUMEROUS PREDEFINED ATTRIBUTES WHICH PROVIDE INFORMATION ABOUT DATA TYPES THAT A PROGRAMMER MAY USE

SYNTAX:

Type\_Name'Attribute\_Identifier [(parameter)]

# INSTRUCTOR NOTES

ATTRIBUTES ALLOW FOR MORE READABLE/MAINTAINABLE CODE BY REFLECTING THE REAL-WORLD STRUCTURE RATHER THAN A SPECIFIC H/W OR APPLICATION DATA VALUES. FOR EXAMPLE X := Real'Digits; IF MOVED TO A NEW MACHINE ONLY NEED TO CHANGE TYPE Real, THE ALGORITHM CAN REMAIN UNCHANGED. ALSO 'DIGITS' REPRESENT OUR CONCEPTUAL VIEW MORE ACCURATELY THEN SAY 8.

# ATTRIBUTES OF NUMERIC TYPES

- T'First - IDENTIFIES SMALLEST VALUE OF THE INTEGER TYPE T
- T'Last - IDENTIFIES LARGEST VALUE OF THE INTEGER TYPE T
- FX'Delta - THE VALUE SPECIFIED IN THE ACCURACY DEFINITION OF A  
FIXED POINT TYPE
- FL'Digits - THE NUMBER OF SIGNIFICANT DIGITS IN A FLOATING POINT  
TYPE

## FOR EXAMPLE:

```
Integer'First YIELDS -32768      -- ON A 16-BIT MACHINE
Integer'Last YIELDS +32767      -- ON A 16-BIT MACHINE
type Real is digits 8;
Real'Digits YIELDS 8
```

INSTRUCTOR NOTES

DON'T EXPLAIN SUBTYPES HERE. JUST STEP THROUGH THE EXAMPLE TO SET UP THE MOTIVATION.

WHAT CAN WE DO? ...

AS YOU GO THROUGH THE EXAMPLE ASK WHAT ARE THE TYPES USED AND WHAT TYPE THE OBJECTS ARE.



## SUBTYPES - MOTIVATION

### CONTEXT:

```
type Line_Length_Type is range 0 .. 1000;  
type Word_Length_Type is range 0 .. 30;  
Line_Length : Line_Length_Type;  
Word_Length : Word_Length_Type;
```

### EXAMPLE:

```
Line_Length := Line_Length + Word_Length;  
  
-- **ILLEGAL: Line_Length  
-- and Word_Length are of  
-- different types yet it  
-- makes 'sense' to mix these  
-- objects
```

INSTRUCTOR NOTES

IF WE CREATE A NEW TYPE, THEN WE CAN'T FREELY MIX OBJECTS OF THAT TYPE. BUT IF WE HAVE A SUBTYPE, WE CAN RESTRICT THE VALUES BUT STILL BE ABLE TO MIX THE OBJECTS SINCE THEY ARE NOT A NEW TYPE.

## SUBTYPES - MOTIVATION

WE CAN RESTRICT THE RANGE WITHOUT INTRODUCING NEW TYPES

CONTEXT:

```
type Line_Length_Type is range 0 .. 1000;  
Line_Length : Line_Length_Type;  
subtype Word_Length_Subtype is Line_Length_Type range 0 .. 30;  
Word_Length : Word_Length_Subtype;
```

EXAMPLE:

```
Line_Length := Line_Length + Word_Length;    -- OK
```

## INSTRUCTOR NOTES

### VALUES

- SUBSET OF VALUES OF SOME BASE TYPE
- DEFINED BY MEANS OF A CONSTRAINT

### OPERATIONS

- SUBTYPE HAS ALL THE SAME OPERATIONS OF THE BASE TYPE
- INTERMEDIATE RESULTS MAY BE OUTSIDE RANGE

A NOTE TO MANAGERS: DESIGNERS AND PROGRAMMERS NEED PROPER TRAINING TO USE THE CAPABILITIES TO MIRROR THE REAL WORLD IN THE ACTUAL CODE.

# SUBTYPES

- AN ADDITIONAL TOOL TO MODEL THE REAL WORLD

## SYNTAX:

```
subtype Identifier is Type_Name [range Lower_Bound .. Upper_Bound];
```

"Type\_Name" IS THE BASE TYPE

## EXAMPLE:

```
type Altitude_Type is range 0 .. 50_000;  
subtype Cruising_Altitude_Type is Altitude_Type range 15_000 .. 35_000;
```

## VALUES OF THE SUBTYPE:

SUBSET OF THE VALUES OF THE BASE TYPE Altitude\_Type  
(15\_000 TO 35\_000)

## OPERATIONS OF THE SUBTYPE:

SAME AS THE BASE TYPE  
(THOSE OPERATIONS AVAILABLE TO CLASS OF INTEGER TYPES)

INSTRUCTOR NOTES

IN PASSING, NOTE WHAT THE USE OF ATTRIBUTES BUYS US. WE CAN TALK ABOUT "THE LARGEST INTEGER" WITHOUT SPELLING OUT THE EXACT VALUE, WHICH IS MACHINE-DEPENDENT.

## PREDEFINED SUBTYPES

subtype Positive is Integer range 1 .. Integer'Last;

subtype Natural is Integer range 0 .. Integer'Last;

-- Integer'Last is the highest value of type Integer

-- for a given implementation

### EXAMPLES:

Units\_Deployed : Natural range 0 .. 105;

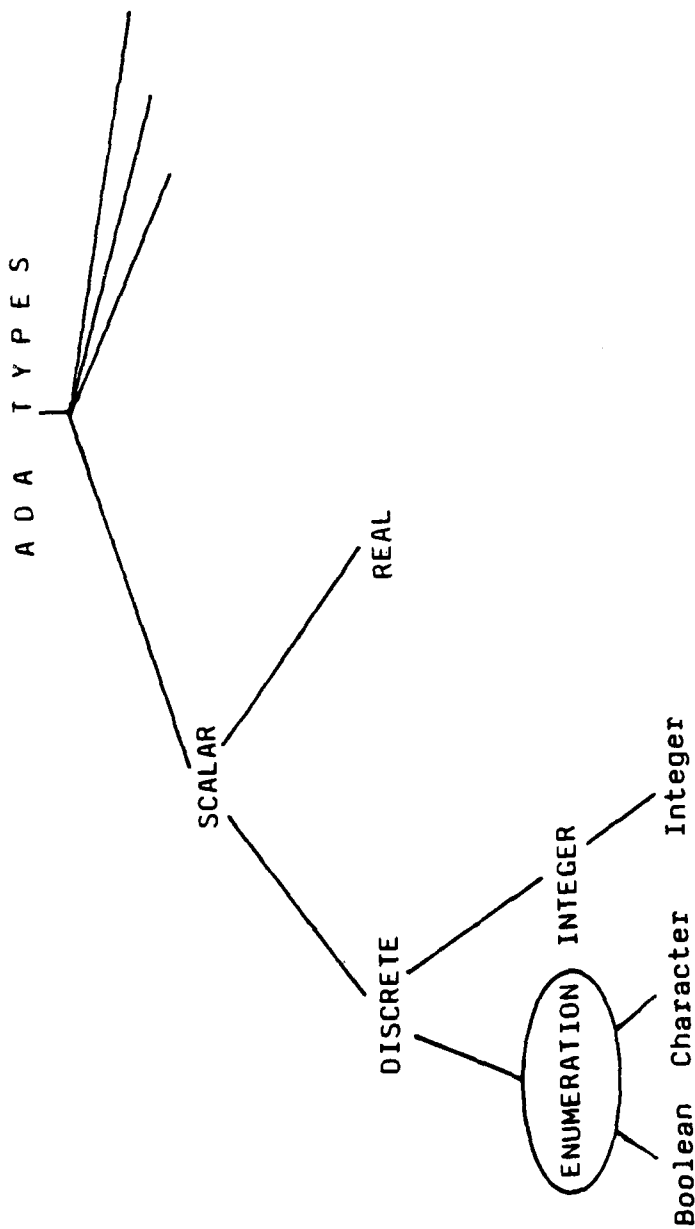
Count : Natural;

## INSTRUCTOR NOTES

USE SLIDE TO FIRST REVIEW WHAT WE HAVE COVERED SO FAR: WHAT ARE TYPES/OBJECTS; WHAT ARE  
NUMERIC TYPES; CONCEPTS AND REASONS FOR ATTRIBUTES AND SUBTYPES (MODELLING OF REAL WORLD  
TO CODE).



# ENUMERATION TYPES



- AN ENUMERATION TYPE IS ONE WHERE THE PROGRAMMER DEFINES BY LISTING, IN ORDER, ALL THE VALUES THAT VARIABLES AND CONSTANTS (I.E. OBJECTS OF THE TYPE) ARE PERMITTED TO ASSUME.

INSTRUCTOR NOTES

IN THE FIRST APPROACH WE ARE FORCED TO REMEMBER DETAILS THAT ARE NOT IMPORTANT AT THIS LEVEL.

IN THE SECOND APPROACH WE DEFINE IN ORDER THE VALUES OBJECTS OF THE TYPE CAN TAKE ON. BUT WE REFLECT THE REAL WORLD SITUATION IN THE VALUES. NOTICE HOW READABLE THIS APPROACH IS.

## WHY ENUMERATION TYPES?

WE HAVE A MONITOR LOCATED IN VARIOUS PARTS OF THE BUILDING, THE LOBBY, THE HALL, THE LAB, THE WORKSHOP, WHICH WE NEED TO TEST IF ACTIVE.

IN MOST LANGUAGES:

WE ARBITRARILY ASSIGN INTEGER VALUES TO EACH LOCATION AND THEN IN A PROGRAM

IF MONITOR = 1 THEN ... (WHERE 1 = LOBBY, 2 = HALL, ETC.)

IN ADA:

WE CAN DIRECTLY EXPRESS THE SITUATION

type Sensor\_Location is (Lobby, Hall, Lab, Workshop);

Monitor : Sensor\_Location;

if Monitor = Lobby then ...

## INSTRUCTOR NOTES

where

- Type\_Name IS A USER SUPPLIED IDENTIFIER FOR THE TYPE BEING INTRODUCED

- Value\_i IS EITHER AN IDENTIFIER OR (AS WE WILL SEE LATER) A CHARACTER LITERAL

NOTE: MUST BEGIN WITH A NON-NUMERIC CHARACTER

I.E. type Baud\_Rate\_Type IS (300, 600, 120); IS ILLEGAL

- THE LIST OF ALLOWED VALUES FOR THE TYPE IS ENCLOSED IN PARENTHESES AND HAS AN IMPLIED ORDER

# ENUMERATION TYPE DECLARATIONS

SYNTAX:

```
type Type_Name is (Value_1, Value_2, ... Value_n);
```

"Value\_1" ARE ENUMERATION LITERALS (IDENTIFIERS OR CHARACTER LITERALS)  
LISTED IN ORDER

EXAMPLES:

```
type Baud_Rate_Type is (B300, B600, B1200);  
type File_Privilege_Type is (Read, Write, Edit, Delete);  
type Day_Type is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

INSTRUCTOR NOTES

IF A VALUE NOT IN THE LIST IS ASSIGNED AN ERROR CONDITION BECOMES KNOWN. WE'LL LOOK AT THIS LATER.

## DECLARATION OF OBJECTS

TYPE DECLARATION: type Baud\_Rate\_Type is (B300, B600, B1200);

OBJECT DECLARATION: Baud\_Rate : Baud\_Rate\_Type;

Baud\_Rate MAY ONLY HAVE VALUES OF B300, B600, OR B1200.

## INSTRUCTOR NOTES

RELATIONAL OPERATORS HAVE MEANING BECAUSE THE ORDER OF ENUMERATION VALUES IS IMPORTANT.  
REMEMBER ENUMERATION TYPES ARE A DISCRETE TYPE WHICH MEANS WE HAVE A KNOWN SUCCESSOR OR  
PREDECESSOR.



# ENUMERATION TYPE OPERATIONS

## ASSIGNMENT

`:=`

## RELATIONAL

`=`      `/=`      `<`      `<=`      `>`      `>=`

## CONTEXT:

`type Baud_Rate_Type (B300, B600, B1200);`

`Baud_Rate : Baud_Rate_Type;`

## EXAMPLES:

`Baud_Rate := B600;`

`if Baud_Rate < B1200 then`

`...`

`end if;`

INSTRUCTOR NOTES

VG 823.1

3-271

# ENUMERATION TYPE ATTRIBUTES

PROVIDE INFORMATION ABOUT THE ORDERING OF THE LIST

- E'First - FIRST VALUE IN THE LISTED VALUES
- E'Last - LAST VALUE IN THE LISTED VALUES
- E'Succ (Value) - LISTED VALUE THAT IMMEDIATELY FOLLOWS (Value)
- E'Pred (Value) - LISTED VALUE THAT IMMEDIATELY PRECEDES (Value)
- E'Pos (Value) - NUMERICAL POSITION OF (Value) IN THE LIST - WHERE  
THE FIRST VALUE HAS POSITION 0
- E'Val (Integer) - ENUMERATION LITERAL AT POSITION (Integer) IN LIST

WHERE E IS THE NAME OF AN ACTUAL ENUMERATION TYPE

INSTRUCTOR NOTES

VG 823.1

3-281

# ENUMERATION TYPE ATTRIBUTE EXAMPLE

## CONTEXT:

type Season\_Type is (Spring, Summer, Fall, Winter);

## EXAMPLES:

Season\_Type'First YIELDS Spring  
Season\_Type'Last YIELDS Winter  
Season\_Type'Succ (Summer) YIELDS Fall  
Season\_Type'Pred (Winter) YIELDS Fall  
Season\_Type'Pos (Summer) YIELDS 1  
Season\_Type'Val (3) YIELDS Winter

## INSTRUCTOR NOTES

- Boolean EXPRESSIONS YIELD A Boolean RESULT (True OR False) WHEN EVALUATED

# BOOLEAN TYPES

- type Boolean IS PREDEFINED BY THE LANGUAGE AS  
type Boolean is (False, True);
- MOST OFTEN FOUND AS CONDITIONS IN if STATEMENTS AND ITERATIVE CONTROL STRUCTURES
- BOOLEAN EXPRESSIONS CAN BE ASSIGNED TO VARIABLES:

## CONTEXT:

```
Year      : Integer;  
Leap_Year : Boolean;      -- Object Declaration of type Boolean
```

## EXAMPLE:

```
Leap_Year := (Year mod 4 = 0 and Year mod 100 /= 0) or Year mod 400 = 0;
```

# INSTRUCTOR NOTES

BEWARE THAT ORDER IS IMPORTANT IN ENUMERATION TYPE. TYPE CHARACTER IS AN ENUMERATION TYPE. THUS IN COMPARING UPPER CASE TO LOWER CASE THERE IS A GAP.

NOTE THE UNPRINTABLE CHARACTERS AT THE TOP.



# PREDEFINED CHARACTER TYPE

type CHARACTER is

<i>(nul,</i>	<i>soh,</i>	<i>stx,</i>	<i>etx,</i>	<i>enq,</i>	<i>ack,</i>	<i>bel,</i>
<i>bs,</i>	<i>ht,</i>	<i>lf,</i>	<i>ff,</i>	<i>cr,</i>	<i>so,</i>	<i>si,</i>
<i>dle,</i>	<i>dcl,</i>	<i>dc2,</i>	<i>dc3,</i>	<i>nak,</i>	<i>syn,</i>	<i>etb,</i>
<i>can,</i>	<i>em,</i>	<i>sub,</i>	<i>esc,</i>	<i>gs,</i>	<i>rs,</i>	<i>us,</i>
' '	'!',	'"',	'#',	'%',	'&',	''',
'(',	')',	'*',	'+',	'-',	'.',	'/',
'0',	'1',	'2',	'3',	'4',	'5',	'6',
'8',	'9',	':',	';',	'<',	'>',	'?',
'@',	'A',	'B',	'C',	'E',	'F',	'G',
'H',	'I',	'J',	'K',	'M',	'N',	'O',
'P',	'Q',	'R',	'S',	'U',	'V',	'W',
'X',	'Y',	'Z',	'[',	']',	'^',	'_',
' ',	'a',	'b',	'c',	'e',	'f',	'g',
'h',	'i',	'j',	'k',	'm',	'n',	'o',
'p',	'q',	'r',	's',	'u',	'v',	'w',
'x',	'y',	'z',	{,	'}',	'~',	del);

## INSTRUCTOR NOTES

### ANOTHER USE:

THE ADVANTAGE OF USING A CHARACTER TYPE FOR ROMAN NUMERALS IS THAT NUMBERS CAN BE WRITTEN AS STRINGS:

"MCCXII"

(MENTION IT IN PASSING; ARRAY TYPES COME LATER.)

- ENUMERATION LITERALS CAN BE EITHER IDENTIFIERS OR CHARACTER LITERALS.
- ANY ENUMERATION TYPE FOR WHICH AT LEAST ONE POSSIBLE ENUMERATION LITERAL IS A CHARACTER LITERAL IS SAID TO BE A CHARACTER TYPE.

# CHARACTER TYPES

- IT IS POSSIBLE TO DEFINE OTHER CHARACTER TYPES IN ADDITION TO THE PREDEFINED TYPE Character.

## EXAMPLES:

type Roman\_Digit\_Type is ('I', 'V', 'X', 'L', 'C', 'D', 'M');

type Security\_Code\_Type is ('U', 'C', 'S', 'T', 'E');

- PRIMARY USE IS TO DEFINE ALTERNATE CHARACTER SETS (E.G., EBCDIC)

## INSTRUCTOR NOTES

ALLOW 5 MINUTES.

### SOLUTION:

```
type Character_Set_Type is (BAUDOT, ASCII, ITA, EBDIC);
Number_Of_Stop_Bits : Natural 1 .. 2;
With_Parity : Boolean;
Even_Parity : Boolean;
type Transmission_Mode_Type is (Asynchronous, Synchronous);
type Message_Priority_Type is (Routine, Priority, Immediate, Flash, ECP, Critical);
```

### OTHER POSSIBILITIES:

```
Number_Of_Stop_Bits : Integer 1 .. 2;
type Parity_Status_Type is (Even, Odd);
```

### KEY POINTS:

1. ADA CAN BE USED TO ACCURATELY REFLECT A GIVEN REAL-WORLD PROBLEM.
2. IMPORTANCE OF CHOICE OF NAMES

# EXERCISE

IN COMMUNICATION SYSTEMS, THE FOLLOWING DATA IS USED:

- CHARACTER SET IN USE - POSSIBILITIES INCLUDE BAUDOT, ASCII, ITA, OR EBCDIC.
- NUMBER OF STOP BITS CAN BE 1 OR 2 BITS IN LENGTH
- PARITY CAN BE EVEN, ODD, OR NOT PRESENT
- ASYNCHRONOUS OR SYNCHRONOUS TRANSMISSION MODES
- PRIORITY OF MESSAGE: ROUTINE, PRIORITY, IMMEDIATE, FLASH, ECP, OR CRITICAL.

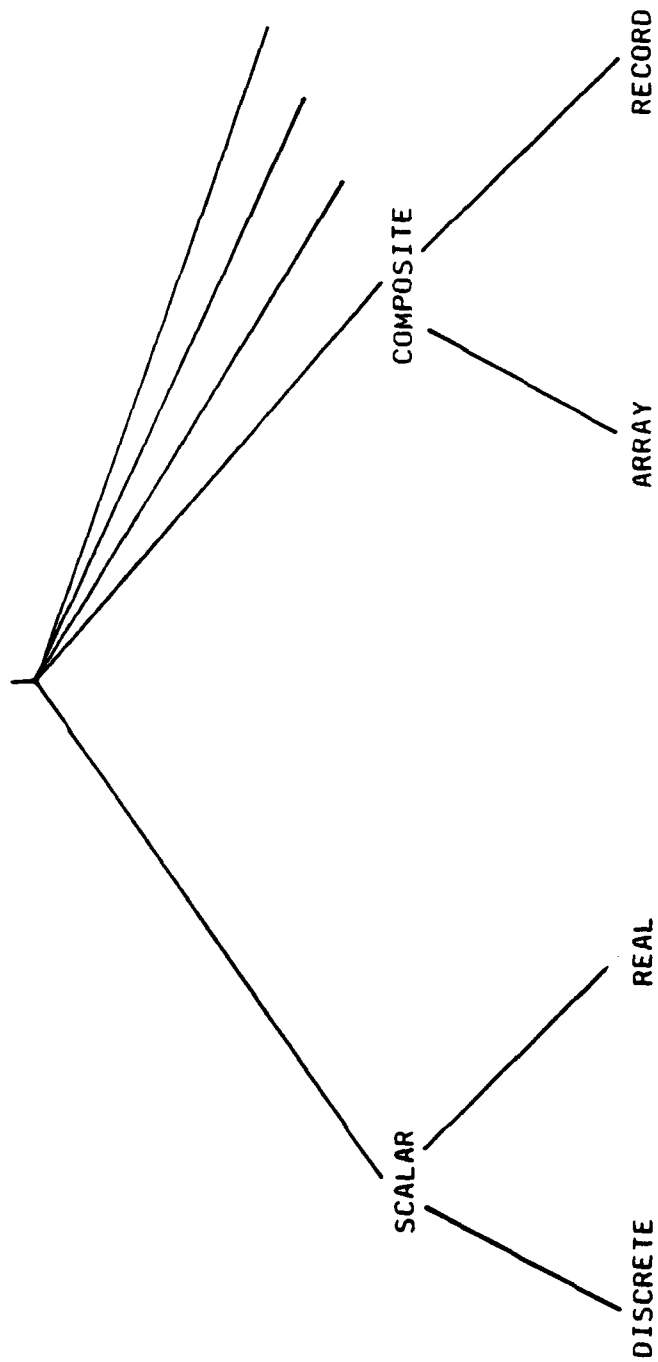
REPRESENT THIS INFORMATION IN ADA.

INSTRUCTOR NOTES

VG 823.1

3-331

# COMPOSITE TYPES



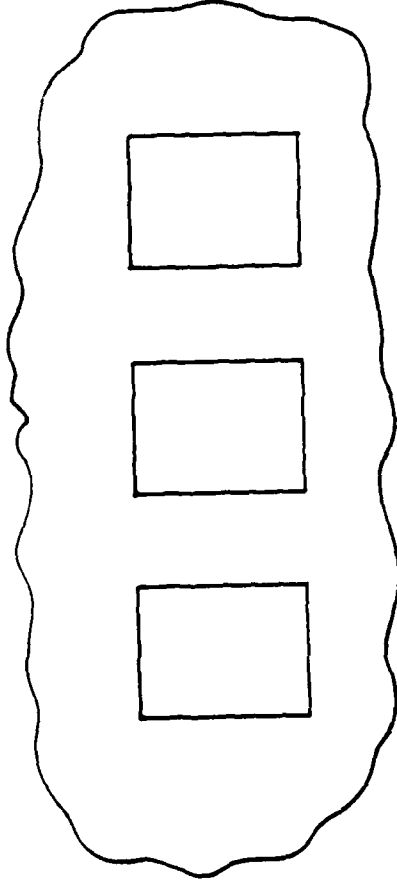
- CANNOT BE DECOMPOSED FURTHER
- BUILDING BLOCKS OF OTHER TYPES
- COMPONENTS MAY BE SCALAR OR COMPOSITE

INSTRUCTOR NOTES

ARRAYS ARE LIKE OTHER LANGUAGES.



# ARRAY TYPE



- COLLECTION OF LIKE OBJECTS SUBJECT TO CERTAIN RULES FOR ACCESSING THESE OBJECTS
- ALL COMPONENTS MUST HAVE THE SAME TYPE

INSTRUCTOR NOTES

VG 823.1

3-351

# ARRAY TYPE

## SYNTAX:

WHAT KIND AND # OF INDICES      TYPE OF COMPONENTS IN ARRAY

type Type\_Name is array (Index\_Subtype { , Index\_Subtype } ) of Component\_Type\_Name;

## TYPE DECLARATION:

type Vector\_Type is array (1 .. 30) of Float;

## OBJECT DECLARATION:

This\_Vector : Vector\_Type;

# INSTRUCTOR NOTES

- INDEX SUBTYPE -- THE SUBTYPE OF THE ARRAY INDEX IN A GIVEN POSITION  
(MAY BE A SUBTYPE OF ANY INTEGER OR ENUMERATION TYPE)
- COMPONENT TYPE -- THE TYPE OF WHAT IS STORED IN THE ARRAY
- DIMENSION(S) -- THE NUMBER OF INDICES

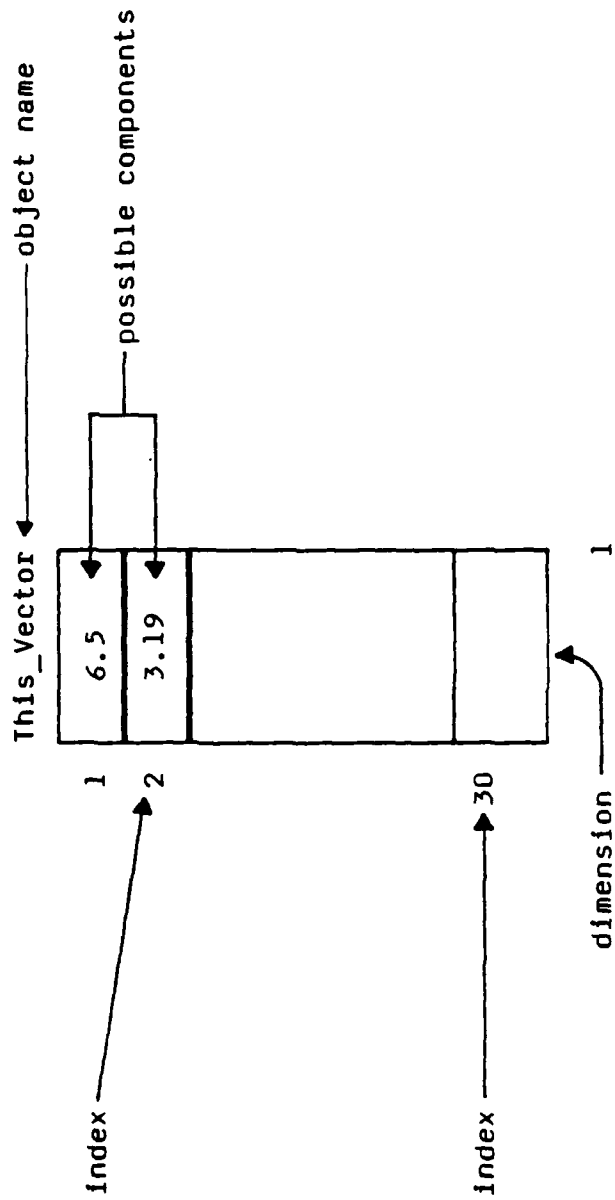
# ARRAY

CONTEXT:

Index Subtype Component\_Type

type Vector\_Type is array (1 .. 30) of Float;

This\_Vector : Vector\_Type;



## INSTRUCTOR NOTES

POINT OUT THROUGH THE EXAMPLES ON THE NEXT TWO SLIDES THE FOLLOWING POINTS:

- COMPONENT CAN BE ANY TYPE (E.G., ARRAY OF ARRAYS)
- INDEX CAN BE ANY DISCRETE TYPE (ENUMERATION OR INTEGER)
- NUMERIC INDICES NEED NOT BE POSITIVE
- RANGE CAN INCLUDE ARBITRARY EXPRESSIONS
- DIFFERENT INDICES CAN BE OF DIFFERENT TYPES
- DIMENSION IS INFERRED FROM THE SPECIFICATION OF THE INDICES
- CAN HAVE AN ARRAY OF ARRAYS

THE FIRST EXAMPLE REPRESENTS A PIXEL OF A GRAPHICS SCREEN USING THE R-G-B METHOD, WHERE 4096 DIFFERENT COLORS ARE MODELED AS A TRIPLE. THIS TRIPLE HOLDS THE "AMOUNTS" OF RED, GREEN AND BLUE THAT CONSTITUTE A PARTICULAR COLOR. IN A GRAPHICS APPLICATION, A TYPICAL SCREEN MIGHT NOW BE DECLARED:

type Screen\_Type is array (0 .. 239, 0 .. 319) of Pixel\_Color\_Type;

## EXAMPLES

### CONTEXT:

```
type Color_Type is (Red, Green, Blue);  
subtype Color_Intensity_Type is Integer range 0 .. 15;
```

### EXAMPLE (INDEXED BY ENUMERATION TYPE):

```
type Pixel_Color_Type is array (Color_Type) of Color_Intensity_Type;
```

### CONTEXT:

```
-- N is either statically defined (e.g. constant, named number)  
-- or dynamically defined (e.g. computed result)
```

### EXAMPLE (INDEX BOUNDS DYNAMICALLY DETERMINED):

```
type Triangular_Matrix_Type is array (1 .. N*(N+1)/2) of Float;
```

INSTRUCTOR NOTES

VG 823.1

3-381



## EXAMPLES (Continued)

### CONTEXT:

type Dial\_Digits is range 0 .. 9;  
type Telephone\_Number\_Type is array (1 .. 10) of Dial\_Digits;

### EXAMPLE (ARRAY OF ARRAYS):

type Active\_Phone\_List is array (1 .. 100) of Telephone\_Number\_Type;

### EXAMPLE (MULTIDIMENSIONAL ARRAYS):

type Developed\_Blocks is array (1 .. 100, 1 .. 10) of Boolean;

## INSTRUCTOR NOTES

ALLOW 5-10 MINUTES FOR THIS EXERCISE. READ THROUGH THE EXERCISE UNDERLINING THE KEY WORDS (E.G., TABLES, POINT SIZE, LETTER, WHITE SPACE ...)

### SOLUTION:

PRESENT SOLUTION TOP DOWN BY RELATING THE SECOND PARAGRAPH TO THE SOLUTION. START AT THE BOTTOM OF THE SOLUTION AND BUILD.

```
subtype Printable_Characters is Character range ' ' .. 'N';  
type Point_Size is range 4 .. 36;  
type White_Space is range 0 .. 72;  
type White_Space_Tables is  
    array (Point_Size, Character) of White_Space;
```

### MEANING:

ADA SUCCINCTLY EXPRESSES THE PROBLEM. TYPES ARE USED TO REPRESENT THE PROBLEM SPECIFICATIONS. AS MANAGERS NEED TO PROMPT THEIR TECHNICAL PEOPLE TO DO THIS.

## EXERCISE

TYPESETTERS MEASURE THE SIZE OF PRINT IN "POINTS," WHERE ONE POINT IS 1/72 INCHES. EXCEPT FOR SPECIAL EFFECTS, TEXT IS GENERALLY PRINTED IN A CONSTANT POINT SIZE. A SOPHISTICATED TYPESETTING PROGRAM MIGHT PLACE AROUND EACH LETTER AN AMOUNT OF WHITE SPACE THAT DEPENDS BOTH ON THE LETTER AND ON THE PREVAILING POINT SIZE.

THE PROGRAM WOULD USE TABLES WHICH, GIVEN A POINT SIZE AND A LETTER, YIELD THE AMOUNT OF WHITE SPACE, MEASURED IN POINTS, TO BE PLACED AROUND THE CHARACTER. ASSUMING THAT THE POINT SIZE CAN VARY BETWEEN 4 AND 36, AND THAT THE MAXIMUM WHITE SPACE EVER ALLOTTED WILL BE ONE INCH (72 POINTS), DECLARE THE TYPES NEEDED TO MANIPULATE SUCH WHITE SPACE TABLES.

INSTRUCTOR NOTES

VG 823.1

3-401

# NOTATION FOR ACCESSING COMPONENTS OF AN ARRAY OBJECT

## SYNTAX

- INDIVIDUAL COMPONENT:  
    Array\_Object (Index\_Value {, Index\_Value } )
- SLICE OF A ONE-DIMENSIONAL ARRAY:  
    Array\_Object (Starting\_Position .. Ending\_Position)  
    Array\_Object (Subtype\_Name)

A SLICE IS A PART OF A ONE-DIMENSIONAL ARRAY, CONSISTING OF CONSECUTIVE COMPONENTS

## CONTEXT FOR EXAMPLES:

```
type Vector_Type is array (1 .. 30) of Float;  
type Coin_Type is (Penny, Nickel, Dime, Quarter, Half_Dollar);  
type Monetary_Value_Type is array (Coin_Type) of Integer;  
Vector_1 : Vector_Type;  
Monetary_Value : Monetary_Value_Type;
```

## EXAMPLES:

```
Vector_1(4)  
Monetary_Value (Penny)  
Monetary_Value (Nickel .. Quarter) -- Slice with three components
```

# INSTRUCTOR NOTES

WHAT CAN WE DO WITH ARRAYS ..

THE KEY POINT: WE CAN WORK WITH THE ARRAY AS A WHOLE, INDIVIDUAL COMPONENT OR SLICES.

THESE OPERATORS ARE SIMILAR TO OTHER LANGUAGES EXCEPT FOR CATENATION. AN EXAMPLE, IF APPROPRIATE FOR CLASS:

```
type Test_Scores_Type is array (1 .. 20) of Integer;  
Old_Score, New_Score: Test_Scores_Type;  
A_New_Score: Integer;  
...  
New_Score := Old_Score (1 .. 19) & A_New_Score;
```

# ARRAY TYPE OPERATIONS

- ASSIGNMENT:  $:=$
- EQUALITY/INEQUALITY:  $=$   $\neq$
- CATENATION:  $\&$  (FOR ONE-DIMENSIONAL ARRAYS ONLY)
- RELATIONAL:  $<$   $\leq$   $>$   $\geq$  (FOR ONE-DIMENSIONAL ARRAYS WITH DISCRETE COMPONENTS ONLY)
- LOGICAL: not xor and or (FOR ONE-DIMENSIONAL ARRAYS WITH Boolean COMPONENTS ONLY)

## INSTRUCTOR NOTES

- AGGREGATE MUST BE COMPLETE
- EACH POSITION MUST BE GIVEN ONLY ONE VALUE
- NOTE: THE NAMED NOTATION FORM WOULD BE UGLY IN THIS CASE AND IS THEREFORE OMITTED.
- POINT OUT THE EXAMPLES OF EACH NAMED NOTATION ALTERNATIVES AND WHAT IT MEANS.

### IF ASKED:

- INSTEAD OF LISTING ALL THE VALUES, THE KEYWORD others MAY BE USED WHERE:
  - IT IS THE LAST CHOICE IN THE LIST; AND
  - ALL THE "others" VALUES ARE THE SAME
- THE others CHOICE CAN BE USED IN EITHER POSITIONAL OR NAMED NOTATION



# ARRAY INITIALIZATION

## AGGREGATES

### CONTEXT:

```
type Dial_Digits is range 0 .. 9;
type Telephone_Number_Type is array (1 .. 10) of Dial_Digits;
type Color_Type is (Red, Green, Blue);
subtype Color_Intensity_Type is Integer range 0 .. 15;
type Pixel_Color_Type is array (Color_Type) of Color_Intensity_Type;
```

### METHODS:

- POSITIONAL (LISTED IN ORDER)
  - Phone\_Number\_1 : Telephone\_Number\_Type := (6, 1, 7, 5, 5, 5, 1, 2, 1, 2);
  - Phone\_Number\_2 : Telephone\_Number\_Type := (9, 1, 4, 1, 1, 1, 1, 1, 1, 1, others => 0);
- NAMED (WITH DISCRETE RANGE)
  - Black\_Pixel : Pixel\_Color\_Type := (Red => 0, Green => 0, Blue => 0);
- NAMED (WITH BAR)
  - Pixel : Pixel\_Color\_Type := (Red | Blue => 15, Green => 2);

NOTE: POSITIONAL AND NAMED MAY NOT BE MIXED.

## INSTRUCTOR NOTES

IF ASKED HOW ARRAY AGGREGATE ASSIGNMENT IS DONE FOR MULTIDIMENSIONAL ARRAY:

CONTEXT:

```
type Entry_Type is (0, X, ' ');
type Tic_Tac_Toe_Type is array (1 .. 3, 1 .. 3) of Entry_Type;
Tic_Tac_Toe_Board : Tic_Tac_Toe_Type := (others => (others => ' '));
```

EXAMPLE:

```
Tic_Tac_Toe_Board := ( 1 => ( X, ' ', ' '),
                        others => (' ', ' ', ' '));
```

# ARRAY ASSIGNMENT

## AGGREGATES

### CONTEXT:

type Coin\_Type is (Penny, Nickel, Dime, Quarter, Half\_Dollar);  
type Monetary\_Value\_Type is array (Coin\_Type) of Integer;  
Monetary\_Value : Monetary\_Value\_Type;

- POSITIONAL NOTATION

### EXAMPLE:

Monetary\_Value := (1, 5, 10, 25, 50);

NOTE: ORDER SIGNIFICANT

- NAMED NOTATION

### EXAMPLE:

Monetary\_Value := (Dime => 10,  
Quarter => 25,  
Penny => 1,  
Half\_Dollar => 50,  
Nickel => 5);

NOTE: ORDER INSIGNIFICANT

INSTRUCTOR NOTES

CONSTRAINED MEANS THE BOUNDS ARE KNOWN AT COMPILE TIME.

VG 823.1

3-441

## CONSTRAINED ARRAY TYPES

- THE KIND OF ARRAY WE'VE BEEN DISCUSSING SO FAR IS A CONSTRAINED ARRAY, IN WHICH THE BOUNDS FOR THE INDEX ARE GIVEN IN THE TYPE DECLARATION (E.G. type List\_Type is  
array (1 .. 15) of Float;)

BUT LET'S SAY WE HAVE A LIST OF SCORES THAT WE WANT TO CALCULATE SOME TEST STATISTICS ON. SOME CLASSES MAY ONLY HAVE 20 STUDENTS; OTHERS, 50; AND SO ON. IN THE SAME PROGRAM WE WOULD WANT TO HAVE SEVERAL LISTS OF SCORES, WHICH ARE OF DIFFERENT LENGTHS BUT NONETHELESS OF THE SAME TYPE.

## INSTRUCTOR NOTES

- ' < > ' MEANS THAT WE DON'T KNOW THE BOUNDS
- BOUNDS FOR ARRAY TYPE DECLARATION MUST BE EITHER ALL CONSTRAINED OR ALL UNCONSTRAINED.  
  
type Not\_Allowed\_Type is array  
    (1 .. 5, Integer range <>) of Integer;   -- \*\*ILLEGAL
- DIFFERENT OBJECTS OF THE TYPE CAN HAVE DIFFERENT BOUNDS
- BOUNDS MUST BE SUPPLIED AT OBJECT DECLARATION TIME WITH AN INDEX CONSTRAINT
- THE TYPE DECLARATIONS MIGHT BE IN A PACKAGE AND THE OBJECT DECLARATIONS COULD BE IN THE SUBPROGRAMS THAT USE THE PACKAGE.

# UNCONSTRAINED ARRAY TYPES

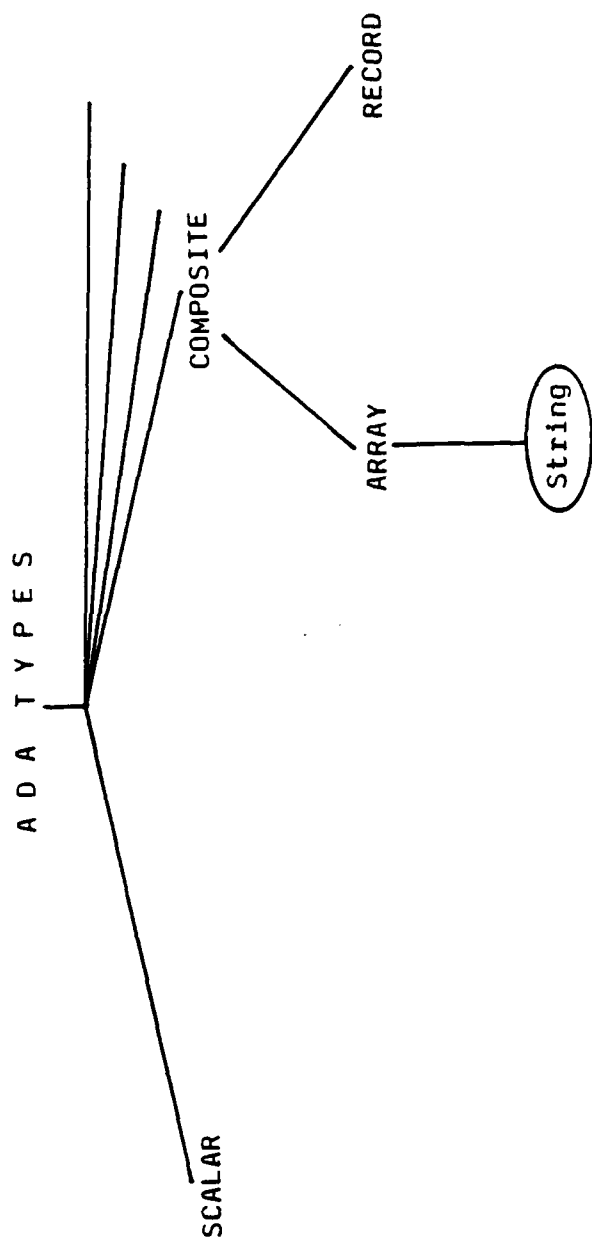
- UNCONSTRAINED ARRAYS LEAVE THE SPECIFICATION OF SPECIFIC BOUNDS TO THE USE OF THE TYPE IN AN OBJECT DECLARATION
- TYPE DECLARATIONS
  - type Unconstrained\_Vector\_Type is array (Integer range <>) of Float;
  - type Matrix\_Type is array (Integer range <>, Integer range <>) of Float;
- OBJECT DECLARATIONS
  - Vector\_1 : Unconstrained\_Vector\_Type (1 .. 5);
  - Vector\_2 : Unconstrained\_Vector\_Type (1 .. 20);
  - Matrix\_1 : Matrix\_Type (-4 .. 0, 1 .. 5);
  - Matrix\_5 : Matrix\_Type (1 .. 5, 1 .. 5);

INSTRUCTOR NOTES

REMIND THE CLASS OF THE OTHER PREDEFINED TYPES WE'VE MET SO FAR.



# THE PREDEFINED TYPE STRING



# INSTRUCTOR NOTES

A CONSTRAINT IS IMPOSED ON THE ARRAY COMPONENT TYPE, STRING.

SUBTYPES CAN BE APPLIED TO ALL TYPES (E.G. ENUMERATION, ARRAYS). THEY ARE NOT LIMITED TO NUMERIC TYPES.

# THE TYPE String

- PREDEFINED UNCONSTRAINED ARRAY TYPE WITH CHARACTER COMPONENTS AND INTEGER INDICES

- "BUILT-IN" DEFINITIONS:

subtype Positive is Integer range 1 .. Integer'Last;  
type String is array (Positive range<>) of Character;

- LOWER BOUND OF INDICES MUST BE 1 OR GREATER.

- POSSIBLE PROGRAMMER-SUPPLIED DEFINITIONS:

subtype Name\_String is String (1 .. 15);  
type Name\_List\_Type is array (1 .. 3) of String (1 .. 8);

- POSSIBLE OBJECT DECLARATIONS:

Name : Name\_String;  
Seminar\_Roster : Name\_List\_Type;  
Social\_Security\_Number : String (1 .. 11);

INSTRUCTOR NOTES

STRING TYPE ATTRIBUTES ARE THE SAME AS ALL ARRAY TYPES.

## ARRAY TYPE ATTRIBUTES

A'FIRST	-	THE LOWER BOUND OF THE INDEX
A'LAST	-	THE UPPER BOUND OF THE INDEX
A'RANGE	-	THE INDEX RANGE (I.E. A'FIRST .. A'LAST)
A'LENGTH	-	THE NUMBER OF VALUES IN THE INDEX

# INSTRUCTOR NOTES

ALLOW 5 MINUTES FOR THIS EXERCISE.

## SOLUTION:

- 1) ID\_Codes\_List : ID\_Codes\_Type (1 .. 50);
- 2) ASSIGNS "B35c" TO THE FIRST ELEMENT OF THE ARRAY OBJECT, "AA1" TO THE SECOND, AND "UNKN" TO ELEMENTS 3, 4 AND 5
- 3) YES
- 4) WHENEVER A '\*' APPEARS AS THE FIRST CHARACTER OF THE FIRST FIVE ELEMENTS OF THE ARRAY OBJECT ID\_Codes\_List, IT REPLACES THE '\*' THE CHARACTER 'Z'

# EXERCISE

GIVEN THE FOLLOWING DECLARATION:

```
type ID_Codes_Type is array (Integer range <>) of String (1 .. 4);
```

1) HOW WOULD WE CREATE A LIST OF 50 ID CODES?

2) WHAT DOES THE FOLLOWING MEAN:

```
ID_Codes_List (10 .. 5) := ("B35c", "AA1", others => "UNKN");
```

3) IS IT TRUE THAT ID\_Codes\_List (1) < ID\_Codes\_List (4)?

4) WHAT DOES THE FOLLOWING CODE EXCERPT DO?

```
for I in 1 .. 5 loop
    if ID_Codes_List (I) (1) = '*' then
        ID_Codes_List (I)(1) := 'Z';
    end if;
end loop;
```

INSTRUCTOR NOTES

VG 823.1

3-501



ADA TYPES

SCALAR

COMPOSITE

ARRAY

RECORD

INSTRUCTOR NOTES

CONTRAST THE COMPONENTS TO ARRAYS.

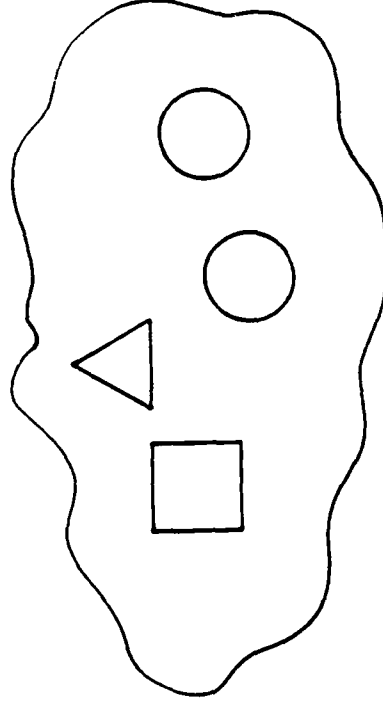
AN EXAMPLE OF LOGICAL RELATIONSHIP: NAME, RANK, NUMBER OF YEARS IN THE SERVICE, CURRENT  
JOB TITLE

VG 823.1

3-51i

# RECORDS

- A COMPOSITE OBJECT CONSISTING OF NAMED COMPONENTS WHICH MAY BE OF DIFFERENT TYPES
- RECORDS SPECIFY LOGICAL RELATIONSHIP, NOT STORAGE LAYOUT



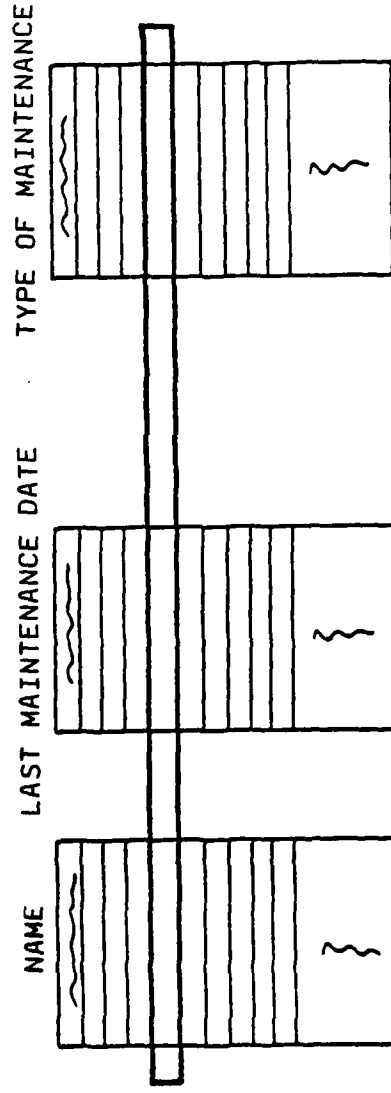
# INSTRUCTOR NOTES

WITH THE ARRAY STRUCTURE, IT IS UP TO THE PROGRAMMER TO KEEP STRAIGHT THAT EACH COMPONENT OF THE PARALLEL ARRAYS GOES WITH A GIVEN PART NAME.

TYPE OF MAINTENANCE MIGHT BE CLEANING, LUBRICATION, REPLACEMENT ... THIS KIND OF INFORMATION, AS WELL AS LAST DATE OF MAINTENANCE, IS TYPICAL OF ENGINE MAINTENANCE (E.G. AIRPLANE ENGINES REQUIRE THAT A LOG BE MAINTAINED AND THAT CERTAIN MAINTENANCE BE DONE EVERY 100 HOURS, ANNUALLY, ETC.)

# ARRAYS VS. RECORDS

- ARRAY STRUCTURE IS LIMITING AS EACH COMPONENT MUST BE OF SAME TYPE.
- ARRAY STRUCTURE FOR PART NAME, LAST MAINTENANCE DATE, TYPE OF MAINTENANCE DONE ...



- ARRAYS PROVIDE A VERTICAL VIEW WHICH DISALLOWS GROUPING OF ONE PART NAME DATA.
- RECORDS ALLOW A HORIZONTAL VIEW, ALLOWS GROUPING OF ONE PART NAME'S DATA.

# INSTRUCTOR NOTES

- COMPONENTS MAY BE OBJECTS OF ANY TYPE
  - IF COMPONENT IS AN ARRAY IT MUST BE FULLY CONSTRAINED.
    - EITHER (1) THE COMPONENT MUST BELONG TO A CONSTRAINED ARRAY TYPE, OR
    - (2) THE COMPONENT MUST HAVE AN INDEX CONSTRAINT.  
(E.G., CAN'T HAVE Job\_Name : String;

# RECORD TYPE DECLARATIONS

EXAMPLE:

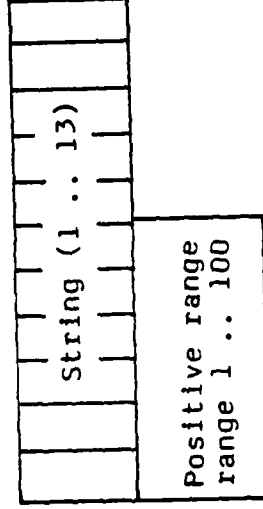
```

type Job_Type is
  record
    Job_Name      : String (1 .. 13);
    Job_Priority  : Positive range 1 .. 100;
  end record;
  
```

- Job\_Type has 2 components

FIRST COMPONENT  $\xrightarrow{\text{named}}$  Job\_Name

SECOND COMPONENT  $\xrightarrow{\text{NAMED}}$  Job\_Priority



# INSTRUCTOR NOTES

RECORD OBJECTS ARE DECLARED LIKE OTHER OBJECTS.

THE INITIAL VALUE IS SUPPLIED BY AN AGGREGATE LIKE WE SAW FOR ARRAYS.

TO ACCESS COMPONENTS OF A RECORD WE USE DOT NOTATION.

RECORD COMPONENTS ARE NOT FIELDS.

A SLIGHTLY MORE SOPHISTICATED SENSOR WILL BE REVISITED IN THE DISCUSSION ON VARIANT RECORDS. IT WILL HANDLE THE PROBLEM OF WHAT DATA VALUES TO USE WHEN THE READING IS INVALID ANYWAY.



# RECORD OBJECT DECLARATION - INITIAL VALUE

## CONTEXT:

```
type Sensor_Type is
  record
    Valid_Reading : Boolean;
    Code          : String (1..13);
    Value         : Float range -50.0 .. 400.0;
  end record;
```

## EXAMPLE:

```
Heat_Sensor : Sensor_Type := (Valid_Reading => True,
                               Code          => "C4H55NW",
                               Value        => 80.5);
```

OR

```
Heat_Sensor : Sensor_Type := (True, "C4H55NW", 80.5); -- POSITIONAL NOTATION
```

## CREATES:

Heat\_Sensor.Valid\_Reading

Heat\_Sensor.Code

Heat\_Sensor.Value

True	Heat_Sensor				
C	4	H	5	5	N
80.5		W			

INSTRUCTOR NOTES

THIS FORMALIZES WHAT WE'VE BEEN LOOKING AT. NOTE THAT RECORD TYPES ARE THE ONLY TYPE  
THAT ALLOWS DEFAULT INITIAL VALUES.

# RECORD TYPE DECLARATION

SYNTAX:

```
type Type_Name is
record
    Component_Name : Component_Type_or_Subtype_Name [:= Default Initial_Value];
    { Component_Name : Component_Type_or_Subtype_Name [:= Default Initial_Value]; }
end record;
```

INSTRUCTOR NOTES

THE INITIALIZATION VALUES IN THE OBJECT DECLARATION OVERRIDES THE DEFAULT VALUES OF THE  
TYPE DECLARATION.

VG 823.1

3-561

# DEFAULT VALUES

## CONTEXT:

```
type Audit_Record_Type is
record
  Num_in_Transit      : Integer := 0;
  Over_flow           : Boolean := False;
  Num_Internal        : Integer := 0;
end record;
```

## EXAMPLES:

Current\_Audit : Audit\_Record\_Type;

## CREATES:

Current_Audit
0
False
0

Past\_Audit : Audit\_Record\_Type := (5, False, 1);

Past_Audit
5
0
1

# INSTRUCTOR NOTES

RECORD AGGREGATE NOTATION IS THE SAME AS ARRAY AGGREGATES. SO DON'T DWELL. IF ASKED:

- others IS RARELY USEFUL FOR RECORDS AS COMPONENTS USUALLY ARE OF DIFFERENT

TYPES:

```
(6, others => 2)          -- ** ILLEGAL
(15, 7, others => 2, Off_Line)  -- ** ILLEGAL
(Site => Off_Line, Fault_Tolerant => False, 0)  -- LEGAL
```

- AN AGGREGATE MUST BE COMPLETE EVEN IF IT SUPPLIES THE SAME VALUES AS THE  
DEFAULT VALUES FOR SOME COMPONENTS.

NAMED NOTATION SHOULD NORMALLY BE USED BECAUSE IT'S EASIER TO READ AND IT'S TOO EASY TO  
WRITE IN THE WRONG ORDER

```
E.G., (3, 2, 1954)      -- FEB. 3, 1954
                        -- OR MARCH 2, 1954
```

VG 823.1

3-571

# RECORD AGGREGATES

- AN AGGREGATE MUST REFER TO A WHOLE RECORD

## CONTEXT:

```
type Site_Type is (On_Line, Off_Line);
type Site_Description_Type
  record
    Available_Lines : Integer range 0 .. 31;
    Secure_Lines    : Integer range 0 .. 7;
    Fault_Tolerant  : Boolean := True;
    Site            : Site_Type;
  end record;
```

```
EUCOM : Site_Description_Type;
```

- POSITIONAL FORM -- LIST IN ORDER

```
if EUCOM = (12, 1, True, On_Line) then ...
```

- NAMED FORM -- MAY BE IN ANY ORDER

```
if EUCOM = (Site => On_Line, Available_Lines => 12, Secure_Lines => 1,
  Fault_Tolerant => True) then ...
```

- MIXED POSITIONAL AND NAMED -- POSITIONAL MUST COME FIRST AND IN ORDER

```
if It_Is = (2, 1, Site => On_Line, Fault_Tolerant => True) then ...
```

## INSTRUCTOR NOTES

### KEY POINTS:

1. RECORDS CAN BE USED AS A WHOLE OR BY THE COMPONENT FOR ASSIGNMENT AND (IN) EQUALITY ONLY.
2. FOR THE OTHER OPERATORS IT IS DONE ON THE COMPONENT ALONE.

DON'T DWELL ON THE EXAMPLES.

THE IDEA OF THE EXAMPLE IS TO DESCRIBE THE ATTITUDE OF PLANES, POSSIBLY INFORMATION AS SUGGESTED BY THE OBJECT DECLARATIONS. A NEW POSITION COULD BE PREDICTED USING THE POWER SETTING, CURRENT ATTITUDE, AND CURRENT POSITION (THIS IS NOT SHOWN HERE). POINT OUT THAT THE DIFFERENT RESTRICTIONS ON THE INTEGER RANGES OF THE COMPONENTS MAKE A RECORD TYPE APPROPRIATE AND ANY ARRAY TYPE INAPPROPRIATE (THE ATTITUDE COMPONENTS ARE MEASURED RELATIVE TO THE STRAIGHT AND LEVEL FLIGHT. ROLL REPRESENTS THE DEGREE TO WHICH THE WINGS ARE BANKED IN A TURN; PITCH REPRESENTS THE DEGREE TO WHICH THE NOSE IS UP OR DOWN AS IN A CLIMB OR A DESCENT; YAW IS THE MOVEMENT OF THE NOSE TO THE LEFT OR RIGHT.)



AD-A165 314

ADA (TRADEMARK) TRAINING CURRICULUM ADA (REGISTERED  
TRADEMARK) FOR SOFTWARE MANAGERS L201 TEACHER'S GUIDE  
VOLUME 1(U) SOFTECH INC WALTHAM MA 1986

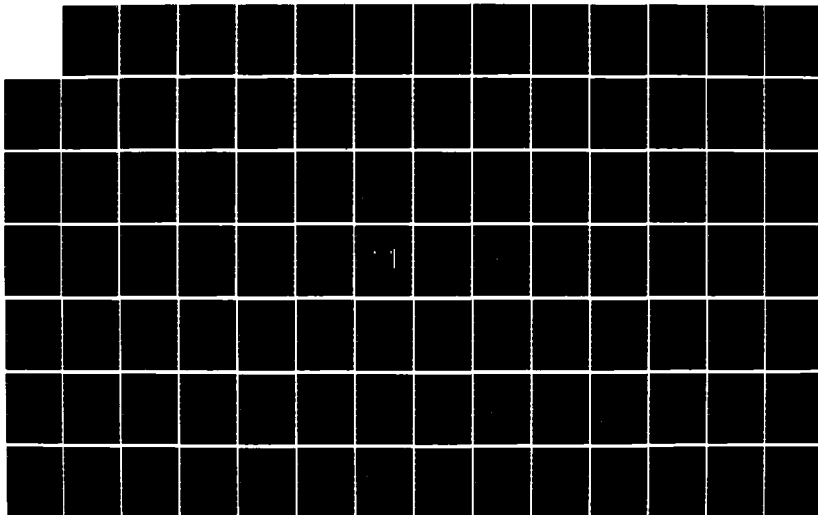
3/5

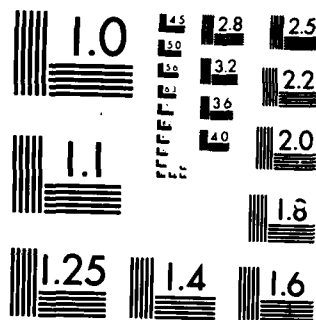
UNCLASSIFIED

DAA807-83-C-K506

F/G 5/9

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

# RECORD OPERATIONS

## CONTEXT:

```
type Attitude_Type is
  record
    Roll : Integer range -90 .. 90;
    Pitch : Integer range -180 .. 180;
    Yaw : Integer range -90 .. 90;
  end record;
```

```
Lead_Plane, Right_Wing_Plane, Left_Wing_Plane : Attitude_Type;
```

## • EQUALITY/INEQUALITY

```
if Lead_Plane = (10, 5, 0) then ...
if Right_Wing_Plane.Pitch = 12 then ...
```

-- WHOLE RECORDS  
-- COMPONENTS OF RECORDS

## • ASSIGNMENT

```
Left_Wing_Plane := (0, 0, 0);
Lead_Plane.Yaw := -4;
```

-- WHOLE RECORD  
-- INDIVIDUAL COMPONENT

## • RELATIONAL, LOGICAL, ARITHMETIC

-- PERFORMED ON INDIVIDUAL COMPONENTS AND NOT ON THE WHOLE RECORD  
(E.G., YOU CANNOT ADD Attitude\_Type OBJECTS)

-- THE OPERATION MUST BE COMPATIBLE WITH THE TYPE OF THE COMPONENT

INSTRUCTOR NOTES

ATTRIBUTES ARE NOT APPLIED TO RECORDS AS A WHOLE.

VG 823.1

3-591

# ATTRIBUTES

- ATTRIBUTES MAY BE APPLIED TO SELECTED COMPONENTS
- ATTRIBUTES MUST BE COMPATIBLE WITH SELECTED COMPONENTS

# INSTRUCTOR NOTES

ALLOW 5 MINUTES.

## SOLUTION:

```
type Liquor_Type is (Scotch, Rum, Gin, Vodka, Rye);  
type Liquor_Manufacturer_Type is (Seagrams, Johnny_Walker, Smirnoff, Bacardi,  
Dewars, Beefeaters, Gordons, Absolute);
```

```
type Inventory_Record_Type is  
record
```

```
    Liquor : Liquor_Type;
```

```
    Manufacturer : Liquor_Manufacturer_Type;
```

```
    Quantity : Float;
```

```
    Price : Float;
```

```
    Inventory_Value : Float;
```

```
end record;
```

ASK THE CLASS HOW THEY WOULD GET AN OBJECT OF THE INVENTORY RECORD (Inventory\_Record :

```
Inventory_Record_Type;)
```

VG 823.1

3-601

# EXERCISE

A LIQUOR STORE NEEDS AN INVENTORY SYSTEM WHICH CAN STORE THE  
FOLLOWING DATA IN A FILE:

- TYPE OF LIQUOR
- MANUFACTURER
- QUANTITY
- UNIT PRICE
- INVENTORY VALUE (QUANTITY \* UNIT PRICE)

THE LIQUOR STORE PURCHASES SCOTCH, RUM, GIN, VODKA, AND RYE FROM  
SEAGRAMS, JOHNNY WALKER, BACARDI, BEEFEATERS, SMIRNOFF, DEWARS,  
GORDONS AND ABSOLUTE.

WRITE THE TYPE DECLARATIONS NEEDED FOR SUCH A SYSTEM

INSTRUCTOR NOTES

EXAMPLE OF STATIC STRUCTURE: Attitude, IT HAS THREE COMPONENTS AND ALWAYS WILL.

VG 823.1

3-611



# PARAMETERIZED RECORDS -- BASIC IDEA

- UP TO NOW RECORDS HAVE HAD STATIC STRUCTURE
- WOULD LIKE RECORDS TO HAVE "PARAMETERS" (CALLED DISCRIMINANTS) WHOSE VALUE AFFECTS STRUCTURE. FOR EXAMPLE:
  - VARY THE COMPONENTS
  - VARY THE SIZE OF AN ARRAY

## INSTRUCTOR NOTES

- INDICATE WHERE THE DISCRIMINANT IS
- DISCRIMINANTS MAY BE USED IN CERTAIN RESTRICTED WAYS IN DECLARATION OF OTHER COMPONENTS
  - AS DEFAULT VALUES, ARRAY BOUNDS, ETC.
- DISCRIMINANTS MAY BE USED TO SELECT COMPONENT
- DISCRIMINANTS ARE COMPONENTS OF THE RECORD AND MUST BE DISCRETE
- DISCRIMINANT MAY BE SUPPLIED WITH DEFAULT VALUES

# USES OF DISCRIMINANTS

- TYPE DECLARATION

type Buffer (Size : Integer) is

record

Position : Integer;

Value : String (1 .. Size); -- VARIABLE LENGTH ARRAY

end record;

- OBJECT DECLARATIONS MUST INCLUDE DISCRIMINANT CONSTRAINT

Small\_Buffer : Buffer (Size => 100);

Large\_Buffer : Buffer (Size => 1000);

INSTRUCTOR NOTES

VG 823.1

3-63i

## VARIANT RECORD -- BASIC IDEA

- CONTAINS A PART THAT VARIES
- RECORD STRUCTURE IS DYNAMICALLY ALLOCATED  
BASED ON THE VALUE OF A DISCRIMINANT

# INSTRUCTOR NOTES

DYNAMICALLY ALLOCATED RECORD STRUCTURE.

GO THROUGH OBJECT DECLARATIONS TO SHOW HOW IT WORKS.

Heat\_Sensor IS NOT CONSTRAINED IN ITS DECLARATION BECAUSE THE DEFAULT CONSTRAINT IS ASSUMED. CONSEQUENTLY, IT CAN HANDLE INVALID READINGS (THROUGH WHOLE RECORD ASSIGNMENT AS SHOWN).

# VARIANT RECORD

CONTEXT:

```

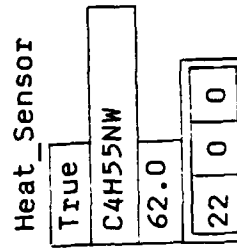
type Time_Type is
  record
    Hour   : Integer range 0 .. 23;
    Minute : Integer range 0 .. 59;
    Second : Integer range 0 .. 59;
  end record;

type Sensor_Type (Valid : Boolean := True) is
  record
    Code: String (1 .. 13);
    case Valid is
      when True =>
        Reading : Float range -50.0 .. 400.0;
        Time     : Time_Type;
      when False =>
        null;
    end case;
  end record;

```

POSSIBLE OBJECT DECLARATIONS:

Heat\_Sensor : Sensor\_Type;



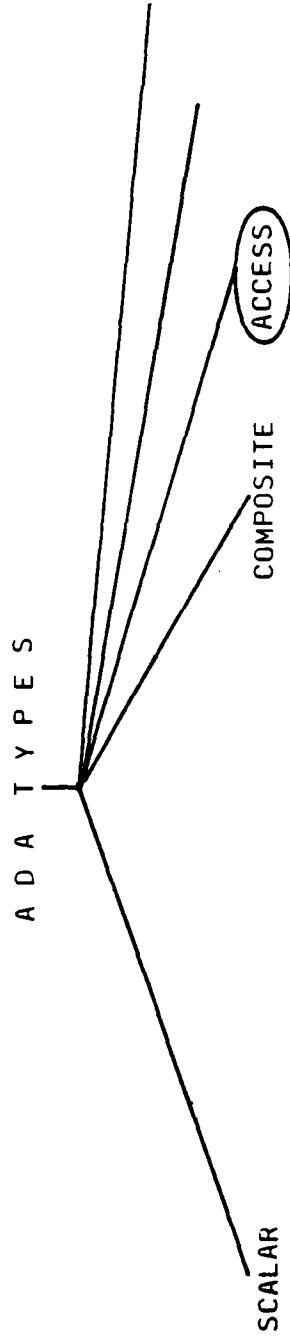
INSTRUCTOR NOTES

VG 823.1

3-651



# ACCESS TYPES



## INSTRUCTOR NOTES

MOTIVATION: NEED TO CREATE OBJECTS DYNAMICALLY. FOR EXAMPLE, A LINKED LIST OF MESSAGES KEPT IN ORDER OF ARRIVAL.

BULLET 3: ONLY THE LINKED DATA STRUCTURES ARE SHOWN IN DETAIL.

EXAMPLE OF SHARING DATA: SUPPOSE YOU HAVE A PERSONNEL FILE SHOWING BOTH MAILING AND HOME ADDRESSES. TYPICALLY THESE ADDRESSES ARE THE SAME. IT IS MORE EFFICIENT AND MORE MAINTAINABLE TO HAVE A SINGLE COPY OF THE ADDRESS POINTED TO AS NECESSARY:

```
type Address_Type is ...;
type Address_Pointer_Type is access Address_Type;
type Personal_Data_Type is
  record
    Name      : String (1 .. 20);
    Home_Address : Address_Pointer_Type;
    Mailing_Address : Address_Pointer_Type;
  end record
```

EXAMPLE OF RAGGED ARRAY: NORMALLY THE ARRAY COMPONENT TYPE MUST BE A CONSTRAINED ARRAY TYPE, MAKING ALL COMPONENTS IDENTICAL LENGTH ARRAYS. POINTERS CAN ALLOCATE UNCONSTRAINED ARRAYS OF ANY LENGTH, SO AN ARRAY OF POINTERS CAN HAVE EACH COMPONENT POINTER POINTING TO DIFFERENT SIZE ARRAYS:

```
type String_Pointer_Type is access String;
type List_Type is array (1 .. 3) of String_Pointer_Type;
L : List_Type := (new String'("oracle", new String'("of"),
                      new String'("Delphi, Greece")));
```

## ACCESS TYPES -- BASIC IDEA

- OBJECTS OF ACCESS TYPES "POINT TO" AN OBJECT
- ACCESS TYPES INVOLVE DYNAMIC CREATION OF OBJECTS.  
(STORAGE RECLAIMED BY SYSTEM WHEN THEY ARE NO LONGER ACCESSIBLE, OR ARE EXPLICITLY DEALLOCATED).
- POSSIBLE USES:
  - AVOID MOVING LARGE AMOUNTS OF DATA
  - IMPLEMENT LINKED DATA STRUCTURES (LISTS, QUEUES, TREES)
  - ALLOW DATA TO BE SHARED
  - IMPLEMENT RAGGED ARRAYS

INSTRUCTOR NOTES

VG 823.1

3-671

# LINKED LIST

CONTEXT:

```

type Node;
type Link is access Node; -- INCOMPLETE TYPE DECLARATION (THE 'THING' TO BE POINTED TO)
type Node is -- ACCESS TYPE TO NODE TYPE
record
  Value : Integer;
  Next : Link;
end record;
Tail, Head, Node_Ptr : Link := null; -- ACCESS OBJECTS WHICH CAN 'POINT'
X : Integer := 2; -- TO TYPE NODE

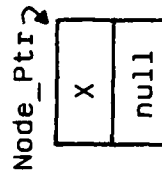
```

TO CREATE A LIST (FIRST NODE):

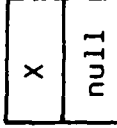
```

Node_Ptr := new Node'(Value => X,
Next => null);

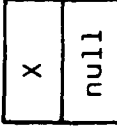
```



Node\_Ptr → Head



Node\_Ptr → Head → Tail



```

Head := Node_Ptr;
Tail := Node_Ptr;

```

INSTRUCTOR NOTES

VG 823.1

3-68i

# LINKED LIST

TO ADD A NODE TO THE END OF THE LIST:

```
Node_Ptr := new Node'(Value => X)
      Next => null);
      -- CREATES NEW NODE AND INITIALIZES

Tail.Next := Node_Ptr;
      -- ADDS NODE TO LIST

Tail := Node_Ptr;
      -- UPDATES THE END OF LIST POINTER, TAIL
```

TO COPY THE CONTENTS OF ONE NODE TO ANOTHER:

```
Node_Ptr.all := Head.all;
```

TO COPY THE CONTENTS OF ONE NODE TO ANOTHER:

```
Node_Ptr.Value := Head.Value;
```

# INSTRUCTOR NOTES

- 1) Head := null;
- 2) Head := Head.Next;
- 3) LET Node\_Ptr BE A POINTER TO THE NEW ELEMENT.  
Node\_Ptr.Next := Head;  
Head := Node\_Ptr;

ALLOW 5 MINUTES FOR THIS.



# EXERCISE

GIVEN A LIST OF INTEGERS DEFINED BY

```
type Node;  
type Link is access Node;  
type Node is  
  record  
    Value : Integer;  
    Next  : Link;  
  end record;  
Head, Tail, Node_ptr : Link;
```

HOW DO YOU:

- 1) INITIALIZE THE LIST TO EMPTY
- 2) DELETE THE FIRST ELEMENT FROM THE LIST
- 3) ADD AN ELEMENT AT THE FRONT OF THE LIST

## INSTRUCTOR NOTES

DERIVED TYPES ARE AN INFREQUENTLY USED ADA FEATURE.

COMPARE DERIVED TYPES TO SUBTYPES. THE VALUES IN A SUBTYPE BELONG TO THE SAME BASE TYPE, WHEREAS VALUES IN DERIVED TYPES BELONG TO DIFFERENT TYPES. VALUES IN A SUBTYPE MAY BE MANIPULATED IN EXPRESSIONS, BUT VALUES IN DIFFERENT DERIVED TYPES MAY NOT WITHOUT EXPLICIT TYPE CONVERSION.

DERIVED TYPES CAN BE USED ON ANY ADA TYPE.

TO MIX OBJECTS OF DERIVED AND PARENT TYPES REQUIRES AN EXPLICIT TYPE CONVERSION.

BULLET 3: A GIVEN ABSTRACT TYPE MAY BE REPRESENTED IN PACKED FORMAT ON DISK AND UNPACKED FORMAT IN CACHE (MAW) MEMORY. BY DERIVING THE DISK VERSION OF THE TYPE, YOU USE PRAGMAS OR REP. SPECS. TO SPECIFY ITS LAYOUT, BUT YOU HAVE STILL INHERITED THE FULL ABSTRACTION (I.E. THE CLASS OF TYPE AND THE OPERATIONS, BOTH PREDEFINED AND USER-DEFINED, USER-DEFINED BEING A SUBPROGRAM WITH A PARAMETER OF THE TYPE) AND WRITE YOUR CODE IN TERMS OF THE ABSTRACT PROPERTIES. YOU USE TYPE CONVERSION TO CHANGE THE PHYSICAL FORMAT OF THE DATA.

# DERIVED TYPES -- BASIC IDEA

OBJECTS CAN HAVE THE SAME STRUCTURE, BUT BE LOGICALLY UNRELATED. DERIVED TYPES PROVIDE A WAY TO REPRESENT THIS ABSTRACTION.

## SYNTAX:

```
type Derived_Type_Name is new Type_Or_Subtype_Name;
```

## FOR EXAMPLE:

```
type Salary_Type is digits range 0.0 .. 100_000.0;  
type Programmer_Salary_Type is new Salary_Type range 0.0 .. 30_000.0;  
Programmer_Salary : Programmer_Salary_Type;  
Salary_Range : Salary_Type;
```

- CREATES A NEW DISTINCT TYPE  
Programmer\_Salary := Salary\_Range + 2\_000.0; -- \*\*ILLEGAL
- OPERATIONS OF NEW TYPE ARE SAME AS THE PARENT TYPE AND INCLUDE TYPE CONVERSION BETWEEN PARENT AND ITS DERIVED TYPES.
- SOMETIMES USEFUL TO MAP SINGLE ABSTRACTION TO DIFFERENT PHYSICAL REPRESENTATIONS (INVOLVES USE OF IMPLEMENTATION DEPENDENT FEATURES)

# INSTRUCTOR NOTES

REMINDE STUDENTS OF THE DIFFERENT CLASSES OF TYPES AND THEIR OPERATIONS:

CLASS OF TYPES	OPERATION			OTHER
	ASSIGNMENT	(IN)EQUALITY	RELATIONAL	
SCALAR	X	X	X	
DISCRETE	X	X	X	
BOOLEAN	X	X	X	BOOLEAN
ENUMERATION	X	X	X	
INTEGER	X	X	X	ARITHMETIC
REAL	X	X	X	ARITHMETIC
COMPOSITE	X	X		
ARRAYS	X	X		
1-DIMENSIONAL	X	X		
			IF SCALAR COMPONENTS	LOGICAL IF BOOLEAN COMPONENTS CATENATION
MULTI-DIM.	X	X		
RECORDS	X	X		

## TYPES -- SUMMARY

- IN ADA, ITEMS OF DATA ARE CALLED OBJECTS
- IN ADDITION TO A VALUE, EVERY OBJECT HAS A PROPERTY CALLED TYPE, WHICH CONSTRAINS THE FORMS THAT THE VALUE MAY TAKE AND THE OPERATIONS THAT THEY MAY BE APPLIED TO THE OBJECT
- THE TYPE OF EACH OBJECT AND THE OBJECT DECLARATION ARE SPECIFIED IN THE DECLARATIVE PART OF A PROGRAM UNIT
- TYPES ALLOW US TO MODEL THE REAL WORLD INTO THE CODE

## INSTRUCTOR NOTES

THIS IS DIFFERENT FROM STRONG TYPING IN PASCAL WHERE TYPE DIFFERENTIATION IS BASED ON THE UNDERLYING REPRESENTATION. IN ADA, YOU CANNOT MIX VALUES OF DIFFERENT INTEGER TYPES IN AN EXPRESSION, LET ALONE ARRAYS, RECORDS, REALS, ENUMERATIONS ...

# STRONG TYPING

- A SET OF RULES THAT FORBID "MIXING APPLES AND ORANGES."

INSTRUCTOR NOTES

AN EXAMPLE OF WHAT CAN HAPPEN WHEN STRONG TYPING IS NOT USED.



# STRONG TYPING TO CORRECT ERRORS BEFORE TESTING

WITHOUT TYPING

CONTEXT:

```
Kilogram_1, Kilogram_2 : Integer;      -- KILOGRAMS  
Pound_1, Pound_2      : Integer;      -- POUNDS
```

EXAMPLE:

```
Kilogram_1 := Kilogram_1 + Kilogram_2;  -- LEGAL AND LOGICAL  
Pound_1    := Pound_1 + Pound_2;       -- LEGAL AND LOGICAL  
Pound_1    := Kilogram_1 + Pound_2;     -- LEGAL BUT NOT LOGICAL INTENT
```

# INSTRUCTOR NOTES

THE SAME EXAMPLE REDONE WITH STRONG TYPING USED. THE COMPILER ENFORCES THE LOGIC. THIS IS THE KEY TO ADA TYPING.

# STRONG TYPING AND ERRORS

## WITH STRONG TYPING

### CONTEXT:

```
type Kilogram is range 0 .. Integer'Last;  
type Pound is range 0 .. Integer'Last;
```

```
Kilogram_1, Kilogram_2 : Kilogram;  
Pound_1, Pound_2      : Pound;
```

### EXAMPLE:

```
Kilogram_1 := Kilogram_1 + Kilogram_2;      -- STILL LEGAL  
Pound_1    := Pound_1 + Pound_2;           -- STILL LEGAL  
Pound_1    := Kilogram_1 + Pound_2;         -- LOGICAL INCONSISTENCY WILL  
                                              -- BE CAUGHT BY THE COMPILER
```

# INSTRUCTOR NOTES

STRONG TYPING IS ONLY USEFUL WHEN IT IS USED, OTHERWISE THERE ARE NO BENEFITS.

THE POINT IS THAT TYPE CHECKING IS PERFORMED ACROSS COMPILE UNIT AND SUBPROGRAM PARAMETERS.

## ANOTHER STRONG TYPING EXAMPLE

procedure Incorrect\_Example is

Counter : Integer;  
Trajectory\_Angle : Float;

...  
function Sin (Angle : in Float)  
return Float is separate;

begin -- Incorrect\_Example

...  
Trajectory\_Angle := Sin (Counter);  
...

end Incorrect\_Example;

-- \*\*ILLEGAL!

• THE ERROR IS DETECTED AT COMPILE TIME RATHER THAN INTEGRATION

INSTRUCTOR NOTES

VG 823.1

3-761

## NOTE

STRONG TYPING CAN ONLY DETECT ERRORS WHEN IT IS USED TO  
DISTINGUISH BETWEEN OBJECTS WITH DIFFERING PROPERTIES

# INSTRUCTOR NOTES

A MESSAGE IN A COMMUNICATION SYSTEM CAN TAKE ON VARIOUS FORMS AND HAVE VARIOUS PRIORITIES. ALL THIS CAN BE REFLECTED IN THE TYPE DECLARATIONS.

(AFTER A WALK THROUGH) - EXERCISE: "Integer range 4 .. 8" OCCURS THREE TIMES. COULD IT BE THAT THOSE THREE RECORD COMPONENTS ARE REALLY OF THE SAME SUBTYPE OR TYPE? OR IS IT JUST A COINCIDENCE?

ANSWER: ALL THREE COMPONENTS IDENTIFY "NODES" OF THE NETWORK. THE DEFINITION SHOULD HAVE BEEN FACTORED INTO A DEFINITION SUCH AS:

type Node\_ID is range 4 .. 8;



# USER-DEFINED TYPES TO DOCUMENT THE DESIGN

```
package Message_Type_Definitions is

type Date is
record
  Julian_Day : Integer range 1 .. 366;
  Year       : Integer range 0 .. 9;
end record;

type Message is (Service, Init, Link, Fail);
type Severity is (Safe, Warning, Error, Fatal);
type Msg (Msg_Type : Message) is
record
  Id       : Integer range 1 .. 100;
  Text    : String (1 .. 200);
  case Msg_Type is
    when Service => Unit_Num : Integer range 4 .. 8;
    when Init   => Time      : Date;
    when Link   => Link_from : Integer range 4 .. 8;
                  Link_to   : Integer range 4 .. 8;
    when Fail   => Degree    : Severity;
  end case;
end record;

subtype Service_Msg is Msg (Service);
subtype Init_Msg   is Msg (Init);
subtype Link_Msg   is Msg (Link);
subtype Fail_Msg   is Msg (Fail);

...

end Message_Type_Definitions;
```

INSTRUCTOR NOTES

AGAIN THE EXAMPLE IS FROM DATA COMMUNICATION. CERTAIN MODES AND FORMS OF TRANSMISSION  
ARE PERMISSIBLE FOR A GIVEN APPLICATION.

# USER-DEFINED TYPES TO EXPRESS DESIGN CONSTRAINTS

```
package Link_Communications is

  type Orientation_Choice      is (Character, Bit);
  type Data_Transmission_Choice is (Asynch, Synch);
  type Baud_Rate_Choice        is (BR_750, BR_1200);
  type Link_is
  record
    Orientation      : Orientation_Choice;
    Data_Transmission : Data_Transmission_Choice;
    Baud_Rate         : Baud_Rate_Choice;
  end record;
  type Number_Bits_in_Character is Natural range 5 .. 8;
  type Number_Stop_Bits         is Natural range 1 .. 2;

  ...

end Link_Communications;
```

## INSTRUCTOR NOTES

SOME EXAMPLES OF CLASSES OF INTEGER TYPES: SERIAL OR IDENTIFICATION NUMBERS, COUNTERS, DEGREES (OF CIRCLE), INSTRUMENT CALIBRATIONS. EACH CLASS SHOULD BE REPRESENTED BY A DISTINCT USER-DEFINED TYPE.

# SOME POTENTIAL PITFALLS - TYPES

- OVERUSE OF THE PREDEFINED TYPES
  - RETAINING A PHYSICAL VIEW OF TYPES
    - USING Integer BECAUSE IT'S 16 BITS
    - USING RECORDS TO SPECIFY PHYSICAL LAYOUT
  - USING Integer INSTEAD OF AN ENUMERATION TYPE
- TOO MANY USER-DEFINED NUMERIC TYPES
  - CAN PRODUCE A PROLIFERATION OF CONVERSIONS
  - CREATING A NEW TYPE WHEN A SUBTYPE IS THE INTENT

INSTRUCTOR NOTES

VG 823.1

3-80i

## POTENTIAL PITFALLS (Continued)

- LIMITING ARRAY INDICES TO INTEGERS WHEN ENUMERATION TYPES (INCLUDING CHARACTER AND BOOLEAN) WOULD MORE CLOSELY REFLECT THE REAL-WORLD SITUATION  
    type Days\_Of\_Week is (Sun, Mon, Tue, Wed, Thurs, Fri, Sat);  
    type Time\_Sheet\_Type is array (Days\_Of\_Week) of Hours\_Worked;  
    rather than  
    type Time\_Sheet\_Type is array (1 .. 7) of Hours\_Worked;  
    where 1 = Sun, 2 = Mon, ...
- USING "PARALLEL" ARRAYS WHEN AN ARRAY OF RECORDS IS INTENDED
- RECORD WHOSE COMPONENT IS AN ARRAY (OR A RECORD) WHOSE COMPONENTS ARE RECORDS (OR ARRAYS) ... BECOMES UNREADABLE

INSTRUCTOR NOTES

VG 823.1

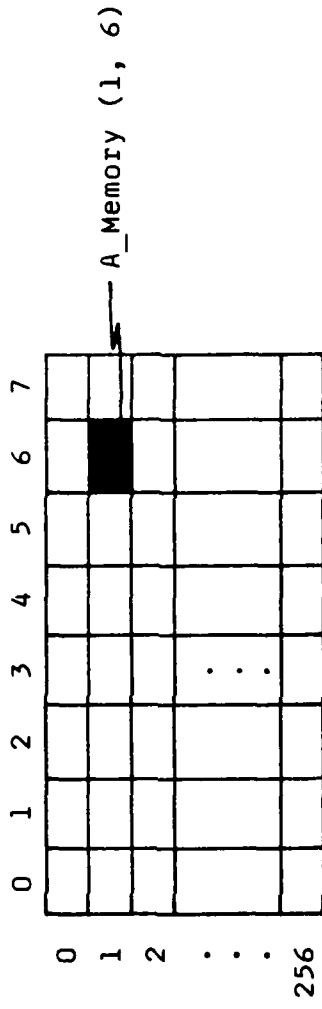
3-811



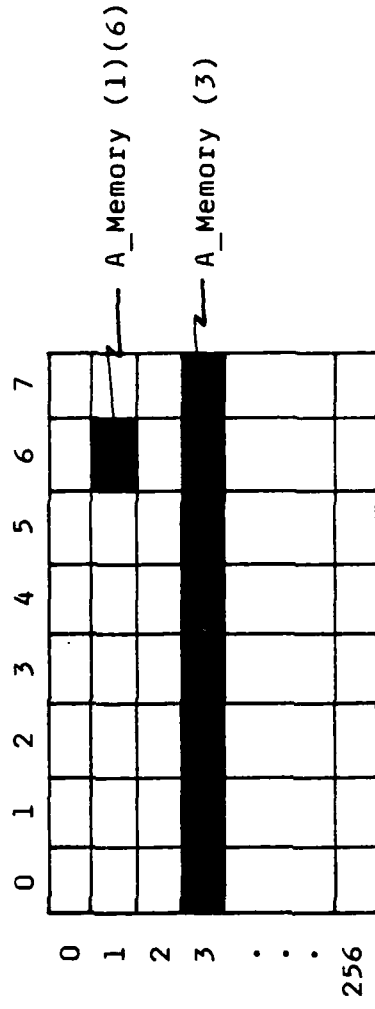
# POTENTIAL PITFALLS (Continued)

- USING TWO-DIMENSIONAL ARRAYS WHEN ARRAY OF ARRAYS IS INTENDED. FOR EXAMPLE:

type Memory\_Type is array (0 .. 256, 0 .. 7) of Boolean;  
 A\_Memory : Memory\_Type;



type Byte is array (0 .. 7) of Boolean;  
 type Memory\_Type is array (0 .. 256) of Byte;  
 A\_Memory : Memory\_Type;



INSTRUCTOR NOTES

VG 823.1

4-i

# **Section 4**

## **STATEMENTS**

VG 823.1

INSTRUCTOR NOTES

VG 823.1

4-11

# STATEMENTS

- PROVIDE A MEANS TO EXPRESS THE ALGORITHMS OF A SOLUTION
- CAN BE USED TO
  - CONTROL LOGIC
  - PERFORM BASIC ACTIONS
  - PERFORM REAL-TIME ACTIONS

## INSTRUCTOR NOTES

WE WILL COVER STATEMENTS FOR REAL-TIME PROCESSING, EXCEPTIONS,  
AND LOW-LEVEL FEATURES LATER WITH THEIR RESPECTIVE ADA FEATURES.

NOW WE'LL CONCENTRATE ON THE UPPER HALF OF THE SLIDE FOR THIS  
SECTION.

# STATEMENTS

FOR BASIC ACTIONS:

ASSIGNMENT  
NULL

FOR FLOW OF CONTROL:

IF  
CASE  
LOOP & EXIT  
GOTO  
PROCEDURE CALLS  
RETURN  
(RAISE -- EXCEPTIONS)

---

FOR REAL-TIME ACTION:

ENTRY CALL  
ACCEPT  
ABORT  
DELAY  
SELECT

FOR EXCEPTIONS:

RAISE  
BLOCK

FOR LOW-LEVEL FEATURES:

CODE

INSTRUCTOR NOTES

EXPRESSIONS CAN BE LITERALS, RESULTS OF ARITHMETIC OPERATIONS, OR FUNCTION CALLS.



# BASIC ACTIONS STATEMENTS

## ASSIGNMENT

- PROVIDES A NEW VALUE TO A VARIABLE

- SYNTAX:

variable\_name := expression;

- EXAMPLES:

Initial\_Value := 0;

Count := Count + 1;

Average := Mean (List\_Of\_Scores);

## NULL

- STATES EXPLICITLY THAT NO ACTION IS TO BE PERFORMED

- SYNTAX:

null;

## INSTRUCTOR NOTES

### POINT OUT:

1. SYNTAX BRIEFLY
2. PROCEDURE CALL EXAMPLES - WE CAN HAVE ZERO OR MORE PARAMETERS
3. FUNCTIONS MUST HAVE A RETURN STATEMENT, PROCEDURES MAY HAVE - WE'LL  
LOOK AT THIS IN GREATER DETAIL IN A LATER SECTION.

# CONTROL STATEMENTS

## PROCEDURE CALLS

- INITIATE EXECUTION OF A PROCEDURE

- SYNTAX:

```
Some_Procedure_Name [(Actual_Parameter | ,Actual_Parameter{ })];
```

- EXAMPLES:

```
Draw_Flag;  
Pop (Element);  
Draw_Square (Start_Point, Length_of_Side);
```

## RETURN

- INDICATES THE TERMINATION POINT OF A SUBPROGRAM

- SYNTAX:

```
return [Some_Expression];
```

- EXAMPLES:

```
return;  
return Sum/Scores_Type (List'Last);
```

INSTRUCTOR NOTES

CONDITIONAL AND ITERATIVE ARE THE STANDARD STRUCTURED PROGRAMMING CONTROL FLOW PATTERNS.

# CONTROL STATEMENTS

## CONDITIONAL

- IF

ALLOWS SELECTION OF A SEQUENCE OF STATEMENTS DEPENDING ON THE VALUE OF A BOOLEAN EXPRESSION

- CASE

SELECTS A SEQUENCE OF STATEMENTS FOR EXECUTION FROM SEVERAL EXCLUSIVE ALTERNATIVES DEPENDING ON THE VALUE OF A DISCRETE EXPRESSION

## ITERATIVE

- LOOP

EXECUTES A PARTICULAR SEQUENCE OF STATEMENTS ZERO OR MORE TIMES

## INSTRUCTOR NOTES

IF STATEMENT IS SIMILAR TO OTHER LANGUAGES SO DON'T DWELL.

POINT OUT:

1. MANDATORY end if
2. THE elsif ALTERNATIVE (CONCEPTUALLY EQUIVALENT TO NESTED ifs)

# CONTROL STATEMENTS - IF

## BASIC IF STATEMENT FORMS

<u>Form 1(a)</u> if <u>Condition</u> then <u>statement(s);</u> end if;	<u>Form 1(b)</u> if <u>Condition</u> then <u>statement(s);</u> else <u>else statement(s);</u> end if;
<u>Form 2(a)</u> if <u>Condition 1</u> then <u>statement(s) 1;</u> elsif <u>Condition 2</u> then <u>statement(s) 2;</u> -- possibly other elsifs end if;	<u>Form 2(b)</u> if <u>Condition 1</u> then <u>statement(s) 1;</u> elsif <u>Condition 2</u> then <u>statement(s) 2;</u> -- possibly other elsifs else <u>else statement(s);</u> end if;

- CONDITION MUST EVALUATE TO A BOOLEAN VALUE
- else ALTERNATIVE MUST BE THE LAST ALTERNATIVE

INSTRUCTOR NOTES

BREEZE THROUGH THE BASIC FORMS 1a AND 1b. READ THROUGH FORM 2a TO ILLUSTRATE THE  
elseif. JUST INDICATE THAT THE elseif ALSO HAS AN else PART AS IN THE BASIC if STATEMENT.



# CONTROL STATEMENTS - IF

## EXAMPLES

<u>Form 1(a)</u> if not Sensor_1_Enabled Enable_Sensor; end if;	<u>Form 1(b)</u> if Top_Sensor = On then Power_Up; else Power_Down; end if;
<u>Form 2(a)</u> if X > Y then Largest := X; elsif X < Y then Largest := Y; elsif X = Y then Put ("They are equal"); end if;	<u>Form 2(b)</u> if Line_Too_Short then Change_Layout; elsif Line_Full then New_Line; Put_(Item); else Put_(Item); end if;

## INSTRUCTOR NOTES

ASK THE STUDENTS WHICH VERSION OF THE CODE THEY WOULD RATHER READ AND MAINTAIN. DO NESTED if's REFLECT THE CONCEPT THAT ONE OF THE THREE ALTERNATIVES ROOT PROCESSING PROCEDURES IS TO BE PERFORMED.

AS AN ASIDE, THIS SIMPLICITY OR CLARITY IS ONE THING TO LOOK FOR IN ADA CODE.

## NESTED IFS

- if STATEMENTS MAY BE NESTED.
- THE elsif ALTERNATIVE MAY ELIMINATE THE NEED FOR NESTED ifs.

FOR EXAMPLE:

```
if Discriminant < 0.0 then
  Process_Complex_Roots;
else
  if Discriminant = 0.0 then
    Process_Two_Idential_Roots;
  else
    Process_Two_Distinct_Roots;
  end if;
end if;

if Discriminant < 0.0 then
  Process_Complex_Roots;
elsif Discriminant = 0.0 then
  Process_Two_Idential_Roots;
else
  Process_Two_Distinct_Roots;
end if;
```

INSTRUCTOR NOTES

VG 823.1

4-9i

# CONTROL STATEMENTS - CASE

## BASIC FORM:

```
case Discrete_Expression is
  when Choice_1 => statement(s);
  when Choice_2 => statement(s);
  .
  .
  .
  when Choice_n => statement(s);
  when others_ => statement(s);  -- OPTIONAL
end case;
```

## SOME RULES:

- Discrete\_Expression MUST EVALUATE TO EITHER AN INTEGER OR ENUMERATION LITERAL.
- THERE MUST BE ONE (AND ONLY ONE) POSSIBLE ALTERNATIVE FOR EACH POSSIBLE VALUE OF THE EXPRESSION
- "when others" REPRESENTS THE CHOICE FOR THOSE VALUES NOT EXPLICITLY LISTED IN THE CHOICES. IF USED, IT MUST APPEAR LAST.

# INSTRUCTOR NOTES

## POINT OUT:

1. Message\_Precedence IS OF TYPE Precedence\_Type
2. Message\_Precedence IS THE DISCRETE EXPRESSION THAT IS THE SELECTOR FOR THE ALTERNATIVE WHEN CLAUSES
3. EACH VALUE THAT Message\_Precedence CAN TAKE ON HAS A when ALTERNATIVE LISTED
4. SEVERAL OF THE Message\_Precedence VALUES WILL SELECT THE SAME STATEMENT FOR EXECUTION (BUT THIS IS NOT EASILY SEEN FROM THE EXISTING FORMAT)

# CASE STATEMENT EXAMPLES

## CONTEXT:

```
type Precedence_Type is (Routine, Priority, Immediate, Flash, ECP, Critical);  
Message_Precedence : Precedence_Type;  
type Channel_Type is (Channel_1, Channel_2, Channel_3);  
procedure Transmit_Message (Over : Channel_Type);
```

## EXAMPLE:

```
case Message_Precedence is  
  when Routine =>  
    Transmit_Message (Over => Channel_3);  
  when Priority .. Flash =>  
    Transmit_Message (Over => Channel_2);  
  when ECP | Critical =>  
    Notify_Operator ("Receiving high-precedence message!");  
    Transmit_Message (Over => Channel_3);  
end case;
```

# INSTRUCTOR NOTES

KEY POINT OF THIS AND NEXT SLIDE: ADA OFFERS SUCCINCT WAYS TO STATE THE WHEN CHOICE ALTERNATIVES (I.E. THEY REFLECT THE REAL WORLD MORE ACCURATELY).

DON'T GO INTO DETAIL WITH THE SLIDES. MAKE THE POINT ABOVE. SHOW THE BAR TO INDICATE OR. INDICATE HOW IT SIMPLIFIES AND EASES THE READING OF THE CODE.

SHOW THE RANGE NOTATION. IT SHOULD BE USED WHEN THERE ARE AT LEAST 3 CONSECUTIVE VALUES FOR WHICH THE SAME SEQUENCE OF STATEMENTS APPLIES.



# ALTERNATIVE NOTATION FOR CASE CHOICES

WHEN THE SAME SEQUENCE OF STATEMENTS IS TO BE EXECUTED FOR MULTIPLE CHOICES

FORMAT:

```
when Choice_M | Choice_N => statement(s);
```

OR

```
when Choice_M .. Choice_N => statement(s);
```

CONTEXT:

```
type Channel_Mode_Type is (Channel_Mode_1, Channel_Mode_2, Channel_Mode_3,  
                           Channel_Mode_4, Channel_Mode_5);
```

EXAMPLE:

```
procedure Select_Correct_Processor (Channel_Mode : in Channel_Mode_Type) is  
  
    procedure Process_Synch is separate;  
    procedure Process_Asynch is separate;  
  
    begin -- Select_Correct_Processor  
  
        case Channel_Mode is  
            when Channel_Mode_1 | Channel_Mode_5 =>  
                Process_Synch;  
            when Channel_Mode_2 | Channel_Mode_3 | Channel_Mode_4 =>  
                Process_Asynch;  
        end case;  
  
    end Select_Correct_Processor;
```

INSTRUCTOR NOTES

VG 823.1

4-121

# CASE CHOICE ALTERNATIVE (Continued)

WHEN THE SAME SEQUENCE OF STATEMENTS IS TO BE EXECUTED FOR ALL THE OTHER CHOICES

FORMAT:

```
when Others => statement(s);
```

EXAMPLE:

```
case Code_Letter is
  when 'U' =>
    Message_Classification := Unclassified;
  when 'R' =>
    Message_Classification := Restricted;
  when 'C' =>
    Message_Classification := Confidential;
  when 'S' =>
    Message_Classification := Secret;
  when 'T' =>
    Message_Classification := Top_Secret;
  when 'A' =>
    Message_Classification := Special_Category;
  when others =>
    Notify_Operator ("Security Mismatch");
end case;
```

INSTRUCTOR NOTES

GO THROUGH THE LOOP SLIDES QUICKLY. THIS IS AN AREA MOST ARE FAMILIAR WITH FROM  
PREVIOUS LANGUAGES.

ADA HAS THREE FORMS OF LOOP STATEMENTS. WE'LL LOOK AT EACH IN TURN.

# CONTROL STRUCTURE - LOOPS

## THREE BASIC FORMS

### SIMPLE LOOP

- REPEATEDLY EXECUTES STATEMENTS AD INFINITUM

### FOR LOOP

- REPEATEDLY EXECUTES STATEMENTS FOR SPECIFIC VALUES OF THE LOOP PARAMETER

### WHILE LOOP

- REPEATEDLY EXECUTES STATEMENTS WHILE SOME CONDITION IS TRUE

INSTRUCTOR NOTES

VG 823.1

4-14i

# FIRST FORM - SIMPLE LOOP

```
loop
-- statement(s)
end loop;
```

- LOOP EXECUTES AD INFINITUM

INSTRUCTOR NOTES

WHY ARE INFINITE LOOPS IN THE LANGUAGE? ANSWER:

"INFINITE" LOOPS ARE COMMON IN EMBEDDED SYSTEMS, WHERE CERTAIN ACTIONS (E.G., POLLING SOME INPUT) MUST BE REPEATED "FOREVER" (UNTIL SOMEBODY PULLS THE PLUG).



## SIMPLE LOOP - EXAMPLE

- STATEMENTS IN LOOP REPEATED AD INFINITUM
- LOOP MAY BE GIVEN A LABEL, CALLED A LOOP IDENTIFIER

procedure Immortality is

--

-- no declarations

--

begin -- Immortality

Breathe:

loop

Inhale;

Exhale;

end loop Breathe;

end Immortality;

-- loop identifier

-- loop identifier repeated

## INSTRUCTOR NOTES

### ONLY IF ASKED:

- RANGE IN REVERSE ALWAYS WRITTEN IN ASCENDING ORDER, BUT LOOP PARAMETER ASSUMES VALUES IN REVERSE

- Lower\_Bound, Upper\_Bound CAN BE ANY EXPRESSION

- IF Lower\_Bound > Upper\_Bound LOOP NOT EXECUTED

- IF Lower\_Bound = Upper\_Bound LOOP EXECUTED ONCE

## SECOND FORM - FOR LOOP

```
for Loop_Parameter in [reverse] Discrete_Range
loop
-- statement(s)
end loop;
```

NOTE: Discrete\_Range CAN BE Specified as Lower\_Bound .. Upper\_Bound,  
Array\_Name'Range, OR Discrete\_Type\_Name

- LOOP PARAMETER IMPLICITLY DECLARED BY THE COMPILER. RANGE OF VALUES ARE SUCCESSIVELY ASSIGNED TO THE PARAMETER IN STEPS OF 1 ON EACH ITERATION
- STATEMENTS EXECUTED ONCE FOR EACH VALUE IN RANGE

# INSTRUCTOR NOTES

List'Range COULD HAVE BEEN WRITTEN 1 .. 15. COULD DISCUSS WHICH IS BETTER WITH CLASS.

POINT OUT THAT USE OF Range ALLOWS ALGORITHMS TO BE WRITTEN INDEPENDENT OF THE LENGTH OF  
ARRAY INDICES. IF List\_Type HAD BEEN AN UNCONSTRAINED ARRAY, NO CHANGE IS REQUIRED WITH  
Mean.

# FOR LOOP - EXAMPLE

## CONTEXT:

```
type Scores is digits 5 range 0.0 .. 1500.0;
subtype Scores_Type is Scores range 0.0 .. 100.0;
type List_Type is array (1 .. 15) of Scores_Type;
```

## EXAMPLE:

```
function Mean (List : in List_Type) return Scores_Type is
    Sum : Scores := 0.0;
begin -- Mean
    for I in List'Range loop
        Sum := Sum + List (I);
    end loop;
    return Sum/Scores_Type(List'Last);    -- type conversion
end Mean;
```

INSTRUCTOR NOTES

RELATE THE BULLETS TO THE EXAMPLE.

VG 823.1

4-18i

# FOR LOOP PARAMETER: RESTRICTIONS

- MAY NOT BE MODIFIED WITHIN THE LOOP

- IS NOT AVAILABLE OUTSIDE THE LOOP

function Mean (List : in List\_Type) return Scores\_Type is

    Sum : Scores := 0.0;

    begin -- Mean

        for I in List'Range loop

            Sum := Sum + List (I);

            I := I + 1;

            -- \*\*ILLEGAL

        end loop;

        return Sum/Scores\_Type(List(I)); -- \*\*ILLEGAL

    end Mean;

INSTRUCTOR NOTES

VG 823.1

4-19i



## THIRD FORM - WHILE LOOP

```
while Boolean_Expression  
loop  
  -- statement(s)  
end loop;
```

- STATEMENTS EXECUTED AS LONG AS Boolean\_Expression IS True
- LOOP TERMINATED WHEN THE Boolean\_Expression IS False

INSTRUCTOR NOTES

THIS CODE IS PART OF THE Sqrt FUNCTION SEEN IN SECTION 1.

# WHILE LOOP - EXAMPLE

## CONTEXT:

```
X      : Float;  
Epsilon : constant := 0.000001;  
Root    : Float := 1.0;
```

## EXAMPLE:

```
while abs (X/Root**2 - 1.0) >= Epsilon  
loop  
    Root := (X/Root + Root)/2.0; -- calculates square root of X  
end loop;
```

INSTRUCTOR NOTES

VG 823.1

4-21i

# LOOP WITH EXIT

- EXIT STATEMENT MAY BE USED TO EXPLICITLY EXIT A LOOP

- EXIT TAKES VARIOUS FORMS

```
exit;                                -- unconditional exit
exit when Boolean_Expression;        -- conditional exit
exit Loop_Name;                      -- unconditional exit
exit Loop_Name when Boolean_Expression; -- conditional exit
```

## INSTRUCTOR NOTES

THE CONVENTION USED HERE IS TO ALIGN THE EXIT WITH THE LOOP AND end loop SO IT IS EASY FOR A MAINTENANCE PROGRAMMER (OR READER) OF THE CODE TO SEE HOW THE LOOP CAN TERMINATE.

# USE OF THE KEYWORD exit

IF THE KEYWORD exit IS ENCOUNTERED INSIDE A LOOP THE LOOP TERMINATES AND CONTROL PASSES TO THE POINT IMMEDIATELY AFTER THE APPROPRIATE "end loop".

## CONTEXT:

```
type Command_Type is (About_Face, Right, Left, Forward_March, Halt);  
Command : Command_Type;
```

## EXAMPLE:

```
loop  
  Get (Command);  
  exit when Command = Halt;  
  Process (Command);  
end loop;
```

INSTRUCTOR NOTES

VG 823.1

4-231



# CONTROL STATEMENTS

GOTO

- CAUSES EXPLICIT TRANSFER OF CONTROL

SYNTAX:

goto Some\_Label;

WHERE A LABEL HAS THE FORM:

<< Some\_Label >>

EXAMPLE:

```
for Index in 1 .. 3 loop
...
  goto Continue;
...
end loop;
<< Continue >> Put ("Not here");
```

INSTRUCTOR NOTES

A GOOD PROGRAM CAN BE READ TOP-DOWN.

AD-A165 314

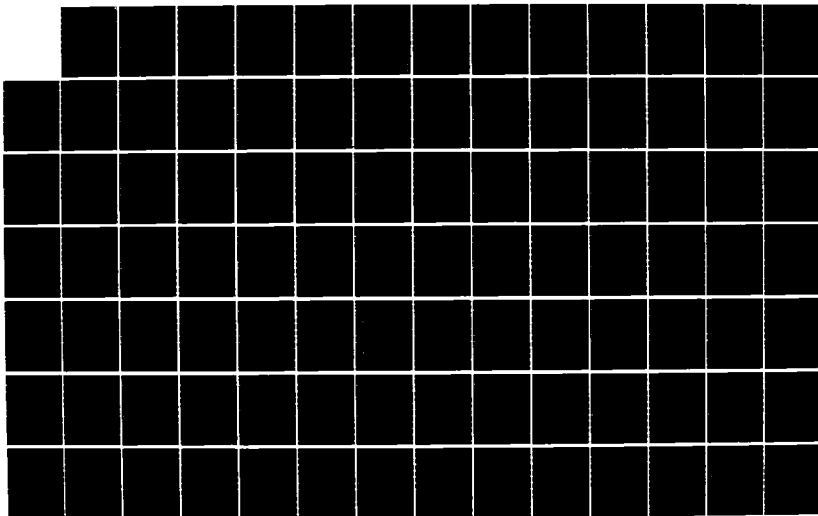
ADA (TRADEMARK) TRAINING CURRICULUM ADA (REGISTERED  
TRADEMARK) FOR SOFTWARE MANAGERS L201 TEACHER'S GUIDE  
VOLUME 1(U) SOFTECH INC WALTHAM MA 1986  
DADB07-83-C-K506

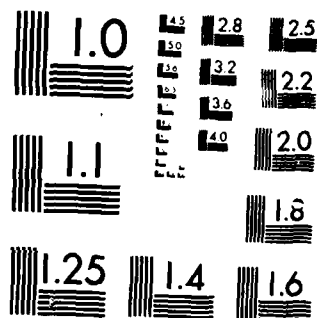
4/5

UNCLASSIFIED

F/G 5/9

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS 1963-A

# SOME POTENTIAL PITFALLS - CONTROL STATEMENTS

- LIMITING LOOP BOUNDS TO INTEGER RANGES OR NOT USING ARRAY ATTRIBUTES AS

LOOP BOUNDS. FOR EXAMPLE:

```
type Hours_Worked is range 0 .. 24;
type Time_Sheet_Type is array (1 .. 7) of Hours_Worked;
Time_Sheet : Time_Sheet_Type;

for Index in 1 .. 7 loop
    Get (Time_Sheet (Index));
end loop;
```

RATHER THAN

```
type Hours_Worked is range 0 .. 24;
type Days_Of_Week is (Mon, Tue, Wed, Thur, Fri, Sat, Sun);
type Time_Sheet_Type is array (Days_Of_Week) of Hours_Worked;
Time_Sheet : Time_Sheet_Type;

for Index in Time_Sheet'Range loop
    Get (Time_Sheet (Index));
end loop;
```

INSTRUCTOR NOTES

POINT OUT THE STYLE CHANGE: VERSIONS ON THE LEFT LOOK LIKE FORTRAN BUT IN ADA SYNTAX.

# PITFALLS (Continued)

- UNDERUSE OF EXPRESSIONS

## Clumsy

```
Result := A + B;
Result := Result / C;
return Result;
```

## Simpler

```
return (A + B) / C;
```

```
if A > B then
  Greater := True;
else
  Greater := False;
end if;
```

```
Greater := A > B;
```

- UNDERUSE OF FUNCTIONS

## Clumsy

```
Read_Temperature (Temp);
if Temp > 68.0 then ...
```

## Simpler

```
if Temperature > 68.0 then ...
-- Where Temperature is
-- a function
```

INSTRUCTOR NOTES

IN GOOD PROGRAMS, goto's ARE RARE (ORDERS OF MAGNITUDE: 1% OF THE ROUTINES MIGHT HAVE ONE goto).

VG 823.1

4-261



## PITFALLS (Continued)

- USING REPEATED IFs INSTEAD OF THE elsif OR case
- USING A goto OR USING AN exit AS A goto
- USING A while loop WITH SUCCESSIVE INCREMENT OF ONE INSTEAD OF A FOR LOOP
- (INFREQUENT) EXCESS OF ZEAL IN AVOIDING goto's (CONVOLUTED LOGIC WHERE A SIMPLE goto MIGHT DO)

INSTRUCTOR NOTES

5-1

VG 823.1

# **Section 5**

## **PACKAGES AND PRIVATE TYPES**

VG 823.1

INSTRUCTOR NOTES

BEFORE PROCEEDING, TALK BRIEFLY ABOUT PROGRAM UNITS.

WE WILL TALK IN TURN ABOUT EACH PROGRAM UNIT (PACKAGES, SUBPROGRAMS, TASKS, GENERICS)

# PROGRAM UNITS

FOR MOST LANGUAGES,

A DESIGN MODULE  $\approx$  SUBROUTINE (OR SUBPROGRAM, IN ADA)

IN ADA,

A DESIGN MODULE  $\approx$  AN ADA PROGRAM UNIT = A PACKAGE, OR

A SUBPROGRAM, OR

A TASK, OR

A GENERIC, OR

COMBINATIONS OF THESE

## INSTRUCTOR NOTES

THE FIRST PROGRAM UNIT WE WILL DISCUSS IS THE PACKAGE. THESE ARE SOME BASIC CONCEPTS REGARDING PACKAGES.

REFER BACK TO THE Vector\_Services PACKAGE PRESENTED AT THE BEGINNING OF THE COURSE.

# PACKAGES

- ARE THE BASIC STRUCTURING UNITS OF ADA SYSTEMS
- PHYSICALLY GROUP FUNCTIONALLY RELATED DATA AND/OR PROGRAM UNITS INTO ONE CONTAINER (ENCAPSULATION)
- PROVIDE FOR REUSABLE SOFTWARE COMPONENTS

INSTRUCTOR NOTES

KEEP THESE IN MIND WHILE WE DISCUSS SOME ASPECTS.



# PACKAGES

CAN BE USED FOR

- GROUPING RELATED PROGRAM UNITS
- NAMED COLLECTIONS OF DECLARATIONS
- FINITE STATE MACHINE
- ABSTRACT DATA OBJECT

## INSTRUCTOR NOTES

THE GROUPING SHOULD BE COMPOSED OF SIMILAR OPERATIONS PERFORMED ON A FEW DATA TYPES. THE OPERATIONS CAN BE PHASES OF SOME PROCESS ON THE DATA OR JUST A COLLECTION OF OPERATIONS ALL USEFUL FOR A PARTICULAR TYPE OF DATA LIKE MATH FUNCTIONS.

POINT OUT:

1. WHAT THE SPEC DOES
2. WHAT THE BODY DOES
3. THE OPTIONAL BEGIN PORTION WITHIN THE BODY

AS WE LOOK AT THE EXAMPLE ON THE NEXT SLIDE, WE'LL LOOK AT WHY WE HAVE THIS STRUCTURE.

# PACKAGES

## PURPOSE:

GROUP LOGICALLY RELATED RESOURCES (TYPES, OBJECTS, PROGRAM UNITS) TOGETHER

## FORM:

{	SPECIFICATION	<pre>package Package_Name is   [DESCRIBES WHAT THE PACKAGE DOES (THE INTERFACE)    THIS INFORMATION IS 'VISIBLE' TO (CAN BE    REFERENCED BY) THIS AND OTHER PROGRAM UNITS] end [Package_Name];</pre>
{	BODY	<pre>package body Package_Name is   [DETAILS HOW THE PACKAGE IMPLEMENTS AN ALGORITHM    OR STRUCTURE    THIS INFORMATION IS 'HIDDEN' FROM (CANNOT BE    DIRECTLY REFERENCED BY) OTHER PROGRAM UNITS] begin   initialization statement(s); end [Package_Name];</pre>

## INSTRUCTOR NOTES

INFORMATION HIDING IS WHAT EASES MAINTENANCE HEADACHES. THE SPEC SPECIFIES THE INTERFACE BETWEEN THE PACKAGE AND ITS USERS. WE CAN MAKE CHANGES TO HOW Square\_Root, Exponential, ETC. ARE ACTUALLY IMPLEMENTED AND NOT AFFECT USERS OF THE PACKAGE.

# PACKAGE - EXAMPLE

```

package Math_Package is
    Pi : constant := 3.14159265358979;
    e  : constant := 2.71828182845904;

    subtype Non_Negative_Float is Float range 0.0 .. Float'Last;

    function Square_Root
        (Square : Non_Negative_Float) return Non_Negative_Float;
    function Exponential (Exponent : Float) return Non_Negative_Float;
    function Natural_Logarithm
        (Power : Non_Negative_Float) return Non_Negative_Float;
    function Common_Logarithm
        (Power : Non_Negative_Float) return Non_Negative_Float;

end Math_Package;

```

SPECIFICATION  
-- WHAT IS  
PROVIDED

```

package body Math_Package is
    -- Code for Square_Root, Exponential, Natural_Logarithm,
    -- Common_Logarithm

end Math_Package;

```

BODY  
-- HOW IT IS  
IMPLEMENTED  
IMPLEMENTED

- THE IMPLEMENTATION OF THE RESOURCES IS PROTECTED AND PHYSICALLY HIDDEN FROM USERS OF THE PACKAGE (INFORMATION HIDING)
- RELIABILITY INCREASED BECAUSE INTERFACE (SPECIFICATION) ERRORS CAN BE EASILY DETECTED
- MAINTAINABILITY INCREASED BECAUSE CHANGES TO THE IMPLEMENTATION (BODY) CAN BE DONE WITHOUT AFFECTING USER PROGRAM UNITS (LOCALIZATION OF THE EFFECTS OF CHANGE)

# INSTRUCTOR NOTES

NOTE HOW WE MUST REFER TO THE Common\_Logarithm AND OTHER PROCEDURES IN THIS EXAMPLE.  
THE NEXT SLIDE SHOWS HOW WE CAN SHORTEN THE REFERENCE TO THE PACKAGE RESOURCES.

# USE OF A PACKAGE

A "with" CLAUSE PROVIDES ACCESS TO THE RESOURCES OF A PACKAGE

## SYNTAX:

```
with Package_Name {,Package_Name};
```

## EXAMPLE:

```
with Math_Package, Plotter_Package;  
procedure_Plot_Log_Scale is  
Data_File : Plotter_Package.File_Type;  
X_Coord, Y_Coord : Float;  
begin -- Plot_Log_Scale  
    while not Plotter_Package.End_of_File (Data_File) loop  
        Plotter_Package.Read (X_Coord);  
        Plotter_Package.Read (Y_Coord);  
        Y_Coord := Math_Package.Common_Logarithm (Y_Coord);  
        Plotter_Package.Plot (X_Coord, Y_Coord);  
    end loop;  
end Plot_Log_Scale;
```

INSTRUCTOR NOTES

VG 823.1

5-71



# USE OF A PACKAGE

A "use" CLAUSE ALLOWS DIRECT NAME REFERENCING OF A PACKAGE'S RESOURCES

## SYNTAX:

```
use Package_Name [,Package_Name];
```

## EXAMPLE:

```
with Math_Package, Plotter_Package;  
use Math_Package, Plotter_Package;  
procedure Plot_Log_Scale is  
  Data_File : Plotter_Package.File_Type;  
  X_Coord, Y_Coord : Float;  
  
begin -- Plot_Log_Scale  
  
  while not Plotter_Package.End_of_File (Data_File) loop  
    Plotter_Package.Read (X_Coord);  
    Plotter_Package.Read (Y_Coord);  
    Y_Coord := Math_Package.Common_Logarithm (Y_Coord);  
    Plotter_Package.Plot (X_Coord, Y_Coord);  
  end loop;  
  
end Plot_Log_Scale;
```

NOTE: THE PREFIXES "Math\_Package." AND "Plotter\_Package." ARE STILL  
LEGAL, BUT NO LONGER REQUIRED TO REFERENCE THE PACKAGE RESOURCES.

INSTRUCTOR NOTES

VG 823.1

5-81

# COMPILATION AND PACKAGES

- A PACKAGE GENERALLY HAS A SPECIFICATION AND A CORRESPONDING BODY
- THE SPECIFICATION CAN BE COMPILED SEPARATELY FROM ITS BODY
- THE USING UNIT OF THE PACKAGE RESOURCES NEEDS ONLY THE PACKAGE SPECIFICATION IN THE PROGRAM LIBRARY TO COMPILE, BUT MUST HAVE THE PACKAGE SPECIFICATION AND BODY TO EXECUTE.

INSTRUCTOR NOTES

THINK OF PACKAGES AS MINI SYSTEM LIBRARIES.

VG 823.1

5-91

# ADA IS PACKAGES

- PACKAGES BETTER CAPTURE THE INTUITIVE NOTION OF A "SUBSYSTEM."
- SUBPROGRAMS AND TASKS (SEE LATER) IMPLEMENT ALGORITHMS.

# INSTRUCTOR NOTES

with Simple\_Graphics; use Simple\_Graphics;  
procedure Draw\_Flag is

```
-- This procedure will draw the flag
-- by first drawing the entire flag
-- pole from (1.5, 4.0) to (1.5, -4.0).
-- The top, right side, and bottom
-- of the flag will be drawn next,
-- in the order specified. Finally
-- the two diagonal lines are drawn.

-- This procedure does not label the
-- diagram with the coordinates of the points.
begin -- Draw_Flag

  -- Draw entire flag pole
  Draw_Line ((1.5, -4.0), (1.5, 4.0));

  -- Draw top
  Draw_Line ((1.5, 4.0), (5.5, 4.0));

  -- Draw right side
  Draw_Line ((5.5, 4.0), (5.5, 0.0));

  -- Draw bottom
  Draw_Line ((5.5, 0.0), (1.5, 0.0));

  -- Draw diagonal top left to bottom right
  Draw_Line ((1.5, 4.0), (5.5, 0.0));

  -- Draw diagonal top right to bottom left
  Draw_Line ((5.5, 4.0), (1.5, 0.0));

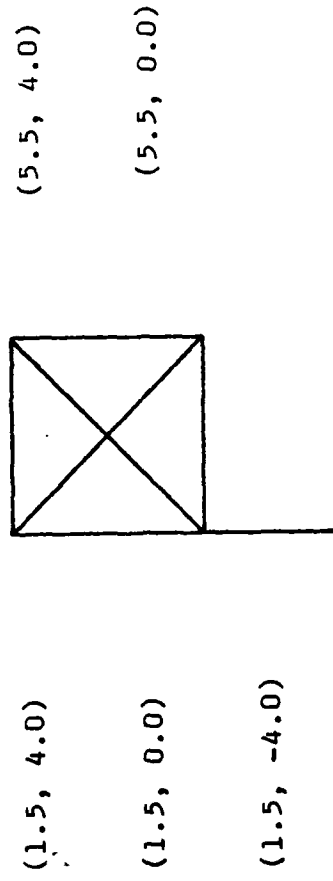
end Draw_Flag;
```

# CLASS EXERCISE

GIVEN THE FOLLOWING PACKAGE SPECIFICATION,

```
package Simple_Graphics is
    type Point_Type is array (1 .. 2) of Float;
    procedure Draw_Line (From : in Point_Type;
                        To : in Point_Type);
    procedure Draw_Circle (Center : in Point_Type;
                        Radius : in Float);
end Simple_Graphics;
```

WRITE A PROCEDURE CALLED Draw\_Flag WHICH DRAWS THE FOLLOWING FLAG. THE CODE SHOULD NOT LABEL THE DIAGRAM WITH THE COORDINATES OF EACH POINT.



INSTRUCTOR NOTES

VG 823.1

5-111



## PACKAGES AS RELATED PROGRAM UNITS

```
package Message_Switch_Services is
    type Message_Type is ...;
    procedure Get_Message (Message : out Message_Type);
    procedure Process_Message (Message : in out Message_Type);
    procedure Send_Message (Message : in Message_Type);
end Message_Switch_Services;

package body Message_Switch_Services is
    -- ACTUAL CODE FOR PROCEDURES
end Message_Switch_Services;
```

- THE PROGRAM UNITS COLLECTIVELY PROVIDE A SERVICE

INSTRUCTOR NOTES

THIS USE OF PACKAGES APPLIES WHEN DATA (VARIABLE OR CONSTANT) IS COMMON TO SEVERAL PROGRAM UNITS.

WHEN PACKAGE CONTAINS ONLY DATA, NO CORRESPONDING PACKAGE BODY IS NEEDED.

INSTRUCTOR CAN USE THE EXAMPLES TO REINFORCE THE DATA TYPES THE STUDENTS HAVE SEEN IN A QUICK QUESTION ANSWER SESSION.

# PACKAGES AS NAMED COLLECTIONS

## OF DECLARATIONS

```
package Channel_Types is
    subtype Channel_Description is String (1 .. 3);
    type Channel_Mode is range 1 .. 5;
    type Char_Set_Type is (Any, ASCII, ITA);
end Channel_Types;

package Metric_Conversions is
    Meters_To_Inches : constant := 39.37;
    Kilometers_To_Miles : constant := 0.62137;
    Liters_To_Quarts : constant := 1.0567;
end Metric_Conversions;
```

## INSTRUCTOR NOTES

FORWARD REFERENCES TO EXCEPTIONS ARE SHOWN IN THE EXAMPLE. JUST STATE THAT THE EXCEPTION DECLARATIONS ARE AS THE NAME IMPLIES -- SPECIFYING EXCEPTIONAL OR ERROR EVENTS -- WHICH WE WILL DISCUSS IN GREATER DETAIL IN A LATER SECTION. IN THIS WAY THE INTUITIVE USE FOR EXCEPTIONS IS BEING BUILT.

A FINITE STATE MACHINE ACCEPTS DATA AND INSTRUCTIONS FROM OUTSIDE AND RETURNS DATA BY SOME PART. IF UNUSUAL EVENTS OCCUR, SOME CONTROL PATHS COMMUNICATE THIS TO THE OUTSIDE WORLD. TO TRANSLATE THIS INTO ADA, THE DATA FLOW IS SPECIFIED BY SUBPROGRAM SPECIFICATIONS WHILE THE CONTROL PATHS ARE TRANSLATED INTO EXCEPTIONS. THE INTERNAL STATES OF THE MACHINE ARE IMPLEMENTED IN THE PACKAGE BODY.

`Vending_Data` IS A PACKAGE THAT MUST BE IMPORTED BECAUSE IT PROVIDES THE TYPE DEFINITIONS FOR THE TYPES `Money`, `Product` AND `Product_Code`.

THE ESSENCE OF THIS STYLE OF DESIGN IS THAT THE VARIABLES THAT IMPLEMENT THE STATE OF THE VENDING MACHINE (`MONEY SUPPLY`, `STOCK LEVEL`, ETC.) ARE "HIDDEN" IN THE PACKAGE BODY. IT IS PHYSICALLY IMPOSSIBLE FOR SOFTWARE OUTSIDE THE PACKAGE TO MODIFY SUCH VARIABLES IN ILLOGICAL WAYS.

# PACKAGES AS FINITE STATE MACHINES

```
with Vending_Data; use Vending_Data; -- TYPE DECL FOR MONEY, PRODUCT, Product_Code
package Vending_Machine is

  -- DATA FLOW OF FSM
  procedure Insert_Change (Coin : in Money);
  procedure Make_Selection (Request : in Product_Code; Item: out Product);
  procedure Make_Change (Coin : in Money; Change : out Money);

  -- COMMUNICATION OF UNUSUAL EVENTS
  Out_of_Change : exception;
  Out_of_Stock : exception;
  Incorrect_Change : exception;

end Vending_Machine;

package body Vending_Machine is

  -- INTERNAL STATES OF FSM IMPLEMENTED

end Vending_Machine;
```

## INSTRUCTOR NOTES

POINT OUT TO MANAGERS: STRUCTURING WITH PACKAGES IS A NEW CONCEPT NOT FOUND IN LANGUAGES MOST PEOPLE ARE FAMILIAR WITH. IT TAKES TRAINING AND EXPERIENCE TO LEARN TO USE THEM EFFECTIVELY.

# SOME PITFALLS - PACKAGES

- NOT USING PACKAGES (TOO MANY "LOOSE" SUBPROGRAMS)

procedure Process(Message)

procedure Get(Message)

procedure Transmit(Message)

- RANDOM COLLECTING OF "THINGS" TO GO IN A PACKAGE (I.E., TREATING PACKAGE LIKE A GARBAGE CAN)

Some\_Package

function Sin (...)

function Mean (...)

procedure Get\_Message (...)

# INSTRUCTOR NOTES

USE OF PACKAGES TO IMPLEMENT "COMPOOLS" OR BLOCKS OF "COMMON" DATA IS POOR PRACTICE. IN GOOD SYSTEMS YOU WILL ALMOST NEVER SEE A VARIABLE DECLARATION IN THE PACKAGE SPEC.

THIS EXAMPLE HAS TYPES FOR DATA DECLARATIONS RELATED TO CHANNEL DESCRIPTIONS, DATE AND TIME, MESSAGE LINE NUMBERS, PARTS OF MESSAGES, PRIORITY AND SECURITY CLASS -- I.E. A HODGE-PODGE.



## PITFALLS (Continued)

### • PACKAGES AS COLLECTIONS OF GLOBAL DATA

```

with ASN; use ASN;
package Global_Types is
  subtype Channel_Description is String (1..3);
  type Channel_Mode is range 1..5;
  type Char_Set_Type is ( EBCDIC, ASCII, IITA );
  type CSN is range 0..999;
  type Ext_Serial_No is
    record
      Number : Serial_Number_Type;
      Extension : Integer range 0..100;
    end record;
  type Julian_Date is range 1..366;
  type Date_Time is
    record
      Date : Julian_Date;
      Time : Duration;
    end record;
  type Logical_Line
  type Part_Name
  type Precedence

  type Security_Classification
    is
      ( Routine,
        Priority,
        Immediate,
        Flash,
        ECP,
        Critical);
    is
      ( Unclassified,
        Encrypted_For_Transmission_Only,
        Restricted,
        Confidential,
        Secret,
        Top_Secret,
        Special_Category,
        DSSCS);

end Global_Types;

```

INSTRUCTOR NOTES

VG 823.1

5-161

# A FURTHER LOOK AT PACKAGES

LET'S CONSIDER COMPLEX NUMBERS ...

```
package Complex_Number is
type Complex_Type is
  record
    Real_Part      : Float;
    Imaginary_Part : Float;
  end record;

function "+" (X, Y : Complex_Type) return Complex_Type;
function "-" (X, Y : Complex_Type) return Complex_Type;
function "*" (X, Y : Complex_Type) return Complex_Type;
function "/" (X, Y : Complex_Type) return Complex_Type;
function Complex (X, Y : Complex_Type) return Complex_Type;
function Real_Of (X : Complex_Type) return Float;
function Imaginary_Of (X : Complex_Type) return Float;

end Complex_Number;
```

• THE PACKAGE ALLOWS THE USER TO SEE AND USE THE DETAILS OF Complex\_Type.

INSTRUCTOR NOTES

VG 823.1

5-171

A USER OF THE PACKAGE COULD DO THE FOLLOWING:

with Complex\_Number; use Complex\_Number;  
procedure Quadratic\_Analysis is

    C : Complex\_Type;  
    D : Complex\_Type;

begin

    ...  
    C.Imaginary\_Part := C.Imaginary\_Part + D.Imaginary\_Part;  
    C.Real\_Part := C.Real\_Part + D.Real\_Part;  
end Quadratic\_Analysis;

- IF THE REPRESENTATION OF Complex\_Type WERE CHANGED TO THE POLAR FORM, THE PROGRAM WOULD NO LONGER BE CORRECT.

## INSTRUCTOR NOTES

IT WOULD BE DESIRABLE NOT TO ALLOW USERS OF PACKAGE Complex\_Numbers TO USE THE ACTUAL REPRESENTATION OF Complex\_Type TO PROMOTE RESISTANCE TO CHANGE. WE CAN ENFORCE THIS ABSTRACT WITH AN ADA FEATURE CALLED PRIVATE TYPES. WE LOOK AT HOW IN A MINUTE, JUST A QUICK REVIEW (IF THEY'VE HAD M101) OF WHAT IS MEANT BY ABSTRACTION. IT MEANS FOCUSING ON THE ESSENTIAL ELEMENTS AND FOR THE MOMENT IGNORING THE UNESSENTIAL DETAILS. WE DO THIS BECAUSE THE HUMAN MIND ONLY HANDLES  $7 \pm 2$  CHUNKS OF INFORMATION AT ANY GIVEN TIME.

IN THIS WAY WE HANDLE THE COMPLEXITY OF THE WORLD AND NO SOFTWARE PROBLEMS.

# PRIVATE TYPES

CAN BE USED TO

- ENFORCE AN ABSTRACTION

VG 823.1

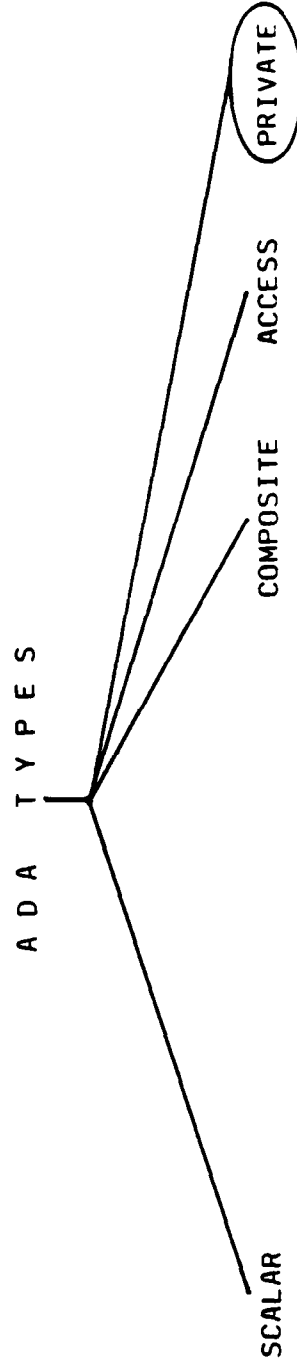
5-18

INSTRUCTOR NOTES

JUST TO REMIND YOU .. PRIVATE TYPES ARE A PART OF THE CLASSES OF ADA TYPES.



# PRIVATE TYPES



INSTRUCTOR NOTES

VG 823.1

5-201

## PRIVATE TYPES: BASIC IDEA

- DECLARE A TYPE BUT CONCEPTUALLY HIDE THE DETAILS OF THE DEFINITION OF THE TYPE FROM THE USER, I.E. PROGRAMMER CANNOT ACCESS THE IMPLEMENTATION
- SUPPORTS DATA ABSTRACTION AND ENCAPSULATION
- PREVENTS THE MISUSE OF INFORMATION OF A TYPE'S INTERNAL REPRESENTATION
- PREVENTS THE CODE FROM BECOMING DEPENDENT UPON AN IMPLEMENTATION AND BECOMING INCORRECT WHEN AN IMPLEMENTATION IS CHANGED

INSTRUCTOR NOTES

VG 823.1

5-211

# PRIVATE TYPES

ARE ONLY DECLARED IN A PACKAGE SPECIFICATION

- SYNTAX:

VISIBLE PART	{	package Package_Name is -- VISIBLE DECLARATION PART type Identifier_Name is [limited] private; private
PRIVATE PART	{	-- PRIVATE DECLARATION PART type Identifier_Name is -- COMPLETE DETAILS OF TYPE DEFINITION end [Package_Name];

INSTRUCTOR NOTES

FUNCTION "+" IS OVERLOADING; WE WILL SEE THIS LATER.

# PACKAGES AS ABSTRACT DATA OBJECTS

- A USER OF Complex\_Number SHOULD NOT NEED TO MANIPULATE OR RELY ON THE IMPLEMENTATION. TO ENFORCE THIS ABSTRACTION:

```
package Complex_Number is
    type Complex_Type is private;

    function "+" (X, Y : Complex_Type) return Complex_Type;
    function "-" (X, Y : Complex_Type) return Complex_Type;
    function "*" (X, Y : Complex_Type) return Complex_Type;
    function "/" (X, Y : Complex_Type) return Complex_Type;
    function Complex (X, Y : Float) return Complex_Type;
    function Real_Of (X : Complex_Type) return Float;
    function Imaginary_Of (X : Complex_Type) return Float;

private
    -- EVERYTHING AFTER THIS IS NOT "VISIBLE" TO PACKAGE USERS
    type Complex_Type is
        record
            Real_Part      : Float;
            Imaginary_Part : Float;
        end record;

end Complex_Number;
```

INSTRUCTOR NOTES

VG 823.1

5-231



# Complex\_Number (Continued)

REFERENCES TO Complex\_Type:

IN THE PACKAGE BODY OF Complex\_Number:

```
X : Complex_Type;  
...  
...X.Real_Part...      -- OK  
...X.Imaginary_Part... -- OK
```

OUTSIDE THE PACKAGE SPEC OR BODY OF Complex\_Number (I.E., USERS OF THE RESOURCE):

```
Y : Complex_Type; -- OK  
...  
...Y.Real_Part...  -- **ILLEGAL!  
Y := (3.0, 5.2)    -- **ILLEGAL!
```

- THE FACT THAT Complex\_Type IS A RECORD IS "HIDDEN" FROM PACKAGE USERS
- THE REPRESENTATION CAN LATER BE CHANGED WITHOUT FEAR THAT ANYTHING OUTSIDE THE PACKAGE WILL BE AFFECTED

INSTRUCTOR NOTES

VG 823.1

5-241

# DATA STRUCTURES AS DATA TYPES

```
package Stack is
    type Stack_Type is private;
    procedure Push (Item : in Integer; On_To : in out Stack_Type);
    function Pop (From : in out Stack_Type) return Integer;
private
    type Stack_Type is
        -- TO USE THE STACK, A USER DOES NOT
        -- NEED TO KNOW THE IMPLEMENTATION
    end Stack;
```

- THUS WE CAN EXTEND THE CONCEPT OF ABSTRACT DATA TYPES TO DATA STRUCTURES

INSTRUCTOR NOTES

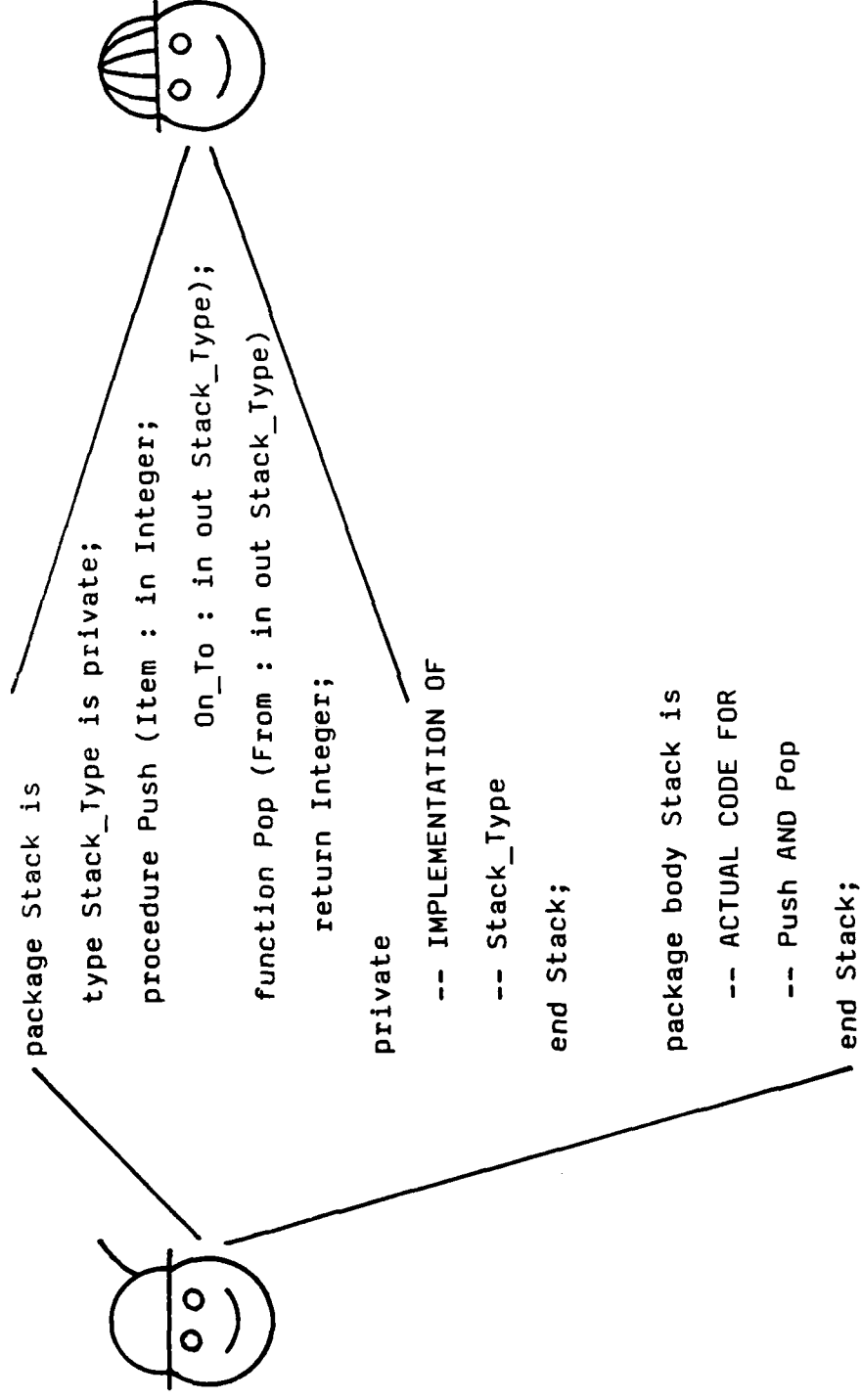
VG 823.1

5-251

# IMPLEMENTOR VS. USER VIEW

IMPLEMENTOR

USER



INSTRUCTOR NOTES

VG 823.1

5-261

# PRIVATE TYPES TO ENFORCE ABSTRACTION

ALLOWS TWO LEVELS OF USER CONTROL OF ABSTRACT DATA TYPES

- PRIVATE TYPES

OPERATIONS AVAILABLE OUTSIDE THE PACKAGE

- ASSIGNMENT
- (IN)EQUALITY
- MEMBERSHIP (IN, NOT IN)
- OPERATIONS (E.G. SUBPROGRAMS) SPECIFIED IN

VISIBLE PART WITH PARAMETERS OF THE PRIVATE TYPE

- LIMITED PRIVATE TYPES

OPERATIONS AVAILABLE OUTSIDE THE PACKAGE

- USER-SPECIFIED OPERATIONS ONLY

INSTRUCTOR NOTES

A SUBPROGRAM USING THIS IO\_PACKAGE CANNOT USE THE INTERNAL REPRESENTATION OF THE  
FILENAME (E.G., ADD 1 TO IT OR COMPARE FILENAMES). IT CAN ONLY Open, Close, Read OR  
Write FILENAMES.

THIS IS AN EXAMPLE OF ENCAPSULATION -- THE ONLY OPERATIONS ARE THOSE GIVEN IN THE  
PACKAGE (THE CONTAINER).



# LIMITED PRIVATE TYPE - EXAMPLE

```
package IO_Package is
  type File_Name is limited private;

  procedure Open  (File : in out File_Name);
  procedure Close (File : in out File_Name);
  procedure Read  (File : in File_Name; Item : out Integer);
  procedure Write (File : in File_Name; Item : in Integer);

private
  type File_Name is
    record
      Internal_Name : Integer := 0;
    end record;
end IO_Package;
```

# INSTRUCTOR NOTES

```
with Stack; use Stack;           -- WE'RE USING STACK PACKAGE

procedure Exercise is
  X, Y : Integer;
  Z    : Float;

  Table_Of_Integers : Stack_Type;

begin -- Exercise

  Push (X, On_To => Table_Of_Integers); -- LEGAL
  Y := Y + Table_Of_Integers(1);        -- ILLEGAL; CAN'T REFER TO IMPLEMENTATION
                                         -- OF Stack_Type OBJECT

  X := Pop (From => Table_Of_Integers); -- LEGAL
  Push (Z, On_To => Table_Of_Integers); -- ILLEGAL, CAN ONLY PUSH OR POP INTEGERS

end Exercise;
```

# CLASS EXERCISE

IDENTIFY WHAT IS LEGAL OR ILLEGAL. (HINT: DON'T WORRY ABOUT SYNTAX)

```
with Stack; use Stack;
procedure Exercise is
  X, Y : Integer;
  Z : Float;
  Table_Of_Integers : Stack_Type;
begin -- Exercise

  Push (X, On_To => Table_Of_Integers);
  Y := Y + Table_Of_Integers(1);
  X := Pop (From => Table_Of_Integers);
  Push (Z, On_To => Table_Of_Integers);

end Exercise;
```

INSTRUCTOR NOTES

NEED TO RESOLVE THE CURRENT LEVEL OF DECOMPOSITION RATHER THEN USE PRIVATE TYPES AS A  
TBD.

# SOME POTENTIAL PITFALLS - PRIVATE TYPES

- PRIVATE DOES NOT MEAN "TBD"
  - USING PRIVATE TYPES TO DEFER DESIGN DECISIONS CAN LEAD TO FORGETTING TO PROVIDE ALL NEEDED FACILITIES FOR THE TYPE IN THE PACKAGE SPEC
- LITERALS FOR Complex\_Number
- HINT: DO NOT TRY TO HIDE INFORMATION THAT IS RELEVANT TO THE USER, E.G.:
    - THE FACT THAT A COMPLEX NUMBER HAS A REAL AND AN IMAGINARY PART
    - THE FACT THAT A FILE HAS A NAME THAT IS AN ASCII STRING
- HIDE HOW THIS INFORMATION IS REPRESENTED:
- ARRAY?
  - RECORD?
  - POINTER?
- PROVIDE FUNCTIONS AND SUBPROGRAMS TO EXTRACT AND MODIFY THE INFORMATION.

INSTRUCTOR NOTES

VG 823.1

6-i

# **Section 6**

## **SUBPROGRAMS**

VG 823.1

INSTRUCTOR NOTES

ADA SUBPROGRAMS ARE A SECOND BUILDING BLOCK OF ADA SYSTEMS. THE FIRST WAS PACKAGES.

CONTRAST SUBPROGRAMS TO PACKAGES (I.E. SUBPROGRAMS ARE EXECUTABLE, PACKAGES ARE LIBRARY  
UNITS THAT MAY CONTAIN SUBPROGRAMS)



# SUBPROGRAMS

- BASIC EXECUTABLE ADA PROGRAM UNIT
- PROVIDE AN ABSTRACT NAME FOR SOME ALGORITHM OR COMPUTATION
- ALLOWS FOR SEPARATION OF THE IMPLEMENTATION OF THE ALGORITHM FROM THE SPECIFICATION OF ITS INTERFACE
- ANALOGOUS TO THE FORTRAN SUBROUTINE AND FUNCTION, BUT MORE RIGOROUS

INSTRUCTOR NOTES

MAIN PROGRAM UNITS GO IN THE PROGRAM LIBRARY.

GIVE AN EXAMPLE OF DEFINITION OF ALGORITHM (E.G. SQRT)

AN EXAMPLE OF OPERATORS FOR ABSTRACT DATA TYPES MIGHT BE PUSH OR POP.

# SUBPROGRAMS

CAN BE USED AS

- MAIN PROGRAM UNITS
- DEFINITIONS OF ALGORITHMS
- DEFINITIONS OF OPERATORS ON ABSTRACT DATA TYPES

INSTRUCTOR NOTES

VG 823.1

6-31

## SUBPROGRAMS

### PROCEDURES

- DEFINES A SEQUENCE OF ACTIONS AND MAY RETURN A VALUE
- IS INVOKED WITH A PROCEDURE CALL STATEMENT

```
Procedure_Name [( Parameter_List )];
```

### FUNCTIONS

- DEFINES A COMPUTATION WHICH MUST RETURN ONE VALUE
- IS INVOKED WITH A FUNCTION CALL WITHIN AN EXPRESSION


```
Function_Name [( Parameter_List )]
```

INSTRUCTOR NOTES

ASSUME Last\_Point, Current\_Point ARE OF Point\_Type; Time IS OF Time\_Type; Velocity IS A  
RETURN PARAMETER.

# PROCEDURE CALL EXAMPLE

```
with Text_IO;  
with Vector_Services; use Vector_Services;  
procedure Compute_Tracking_Data is  
    ...  
begin -- Compute_Tracking_Data  
    ...  
    Calculate_Velocity (Last_Point, Current_Point, Time, Velocity);  
    ...  
end Compute_Tracking_Data;
```

procedure        
call

## INSTRUCTOR NOTES

THIS IS A BETTER STYLE THAN PREVIOUS EXAMPLE.

FUNCTIONS (IN THE MATHEMATICAL SENSE) ARE MUCH EASIER TO REASON ABOUT THAN SEQUENCES OF ACTIONS.

FUNCTION CALLS CAN APPEAR ANYWHERE AN EXPRESSION CAN. FOR EXAMPLE, IN AN ARRAY DECLARATION - ARRAY (1 .. N) ...



# FUNCTION CALL EXAMPLE

```
with Text_IO;  
with Vector_Services; use Vector_Services;  
procedure Compute_Tracking_Data Is
```

```
...
```

```
begin -- Compute_Tracking_Data
```

```
...
```

function call  $\longrightarrow$  Distance := Distance\_Between (Last\_Point, Current\_Point);

```
...
```

```
end Compute_Tracking_Data;
```

## INSTRUCTOR NOTES

INDICATE THE DECLARATIVE AND EXECUTABLE PARTS OF THE BODY.

STRESS AGAIN THAT THE SPECIFICATION IS THE INTERFACE OR CONTRACT BETWEEN THE PROCEDURE AND ITS CALLER; THE BODY OF A SUBPROGRAM IS THE IMPLEMENTATION OF THE ALGORITHM.

# PROCEDURES

## GENERAL FORM OF A PROCEDURE

### SPECIFICATION

```
procedure Procedure_Name [(Parameter_List : [mode] Parameter_Type  
                           ;; Parameter_List : [mode] Parameter_Type))] is  
    -- local declarations (if any)  
begin    -- Procedure_Name  
    -- local statements  
    -- may OPTIONALLY include one or more return statements:  
    [return;]  
    -- to terminate execution of procedure body and return to caller  
end [Procedure_Name];
```

BODY

- PROCEDURES CAN BE A MAIN PROCEDURE, IN A PACKAGE BODY, OR NESTED IN ANOTHER SUBPROGRAM DECLARATIVE REGION

## INSTRUCTOR NOTES

FOR MANAGERS IT IS IMPORTANT TO KNOW IT'S A PROCEDURE AND THE ALGORITHM SHOULD PERFORM THE SERVICE INDICATED BY THE NAME OF THE PROCEDURE. WE'LL LOOK AT THE MEANING OF PARAMETER LISTS SHORTLY. INDICATE WHAT IS THE SPEC AND BODY.

USING THE WORD INTERFACE INSTEAD OF SPEC MAY BE LESS CONFUSING FOR STUDENTS.

POINT OUT THE USE OF THE SHORT CIRCUIT CONTROL FORM and then.

ASSUME THAT Found AND Not\_Found ARE PROCEDURES DECLARED ELSEWHERE IN THIS SCOPE.

# EXAMPLE

```
procedure Search (S : String; C : Character) is
  Position : Integer := S'First;
begin -- Search
  while (Position <= S'Last) and then (S(Position) /= C)
  loop
    Position := Position + 1;
  end loop;
  if Position <= S'Last then
    Found (Where => Position);
  else
    Not_Found;
  end if;
end Search;
```

# INSTRUCTOR NOTES

## POINT OUT:

1. THE DECLARATIVE AND EXECUTABLE REGIONS.
2. "RETURN Type\_Name" SAYS WHAT TYPE (E.G. ARRAY, INTEGER) WILL BE RETURNED
3. "EXPRESSION" MUST BE OF THE TYPE "Type\_Name" NAMED IN THE RETURN STATEMENTS.
4. THE SPECIFICATION AND BODY (AND WHAT EACH DOES)

# FUNCTIONS

## GENERAL FORM OF A FUNCTION

### SPECIFICATION

```
function Function_Name [( Parameter_List : [in] Parameter_Type  
    ); Parameter_List : [in] Parameter_Type{}]] return Type_Name is  
begin  
    -- local declarations (if any)  
    -- Function_Name  
    -- local statements  
    -- MUST include at least one return statement:  
    -- the return statement terminates execution of the subprogram  
    -- and specifies the value to be returned  
    return Expression; -- where Expression is of type Type_Name  
end [Function_Name];
```

BODY

- FUNCTION CAN BE IN PACKAGE BODIES OR NESTED IN ANOTHER SUBPROGRAM DECLARATIVE REGION

# INSTRUCTOR NOTES

## POINT OUT:

1. IT IS A FUNCTION
2. IT RETURNS SOMETHING OF TYPE Float
3. THE RETURN STATEMENT RETURNS Root WHICH IS OF TYPE Float
4. SPEC AND BODY

WE'LL NEXT LOOK AT THE MEANING OF THE PARAMETER LIST.



## EXAMPLE

```
function Sqrt (X : Float) return Float is
    Root : Float;
    ...
begin
    -- Sqrt
    ...
    return Root;
end Sqrt;
```

## INSTRUCTOR NOTES

- BOTH A DECLARATION AND BODY MAY APPEAR FOR A GIVEN SUBPROGRAM. IN THIS CASE THE SPECIFICATION IN THE BODY MUST MATCH THE SPECIFICATION IN THE DECLARATION
- ALTERNATIVELY THE BODY MAY APPEAR ALONE. IN THIS CASE THE BODY SERVES AS BOTH DECLARATION AND BODY, DEFINING BOTH THE INTERFACE WITH THE CALLER AND THE IMPLEMENTATION OF THE PROCEDURE
- NOTE THAT IN A PACKAGE SPECIFICATION, ONLY A SUBPROGRAM DECLARATION IS USED. REMEMBER A SPECIFICATION IS THE INTERFACE, SO TO USE THE PACKAGE WE ONLY NEED THE INFORMATION PROVIDED IN A SUBPROGRAM INTERFACE (I.E. ITS SPECIFICATION)

# PLACEMENT OF SUBPROGRAMS

- EXAMPLE: SUBPROGRAM SPECIFICATION APPEARING IN A SUBPROGRAM DECLARATION.

```
package Vector_Package is
  type Vector_Type is array (Integer range <>) of Integer;
  function Average (V : Vector_Type) return Float;
end Vector_Package;
```

- EXAMPLE: SUBPROGRAM SPECIFICATION APPEARING AS PART OF THE BODY.

```
procedure Process_Sample_Data is
  type Vector_Type is array (Integer range <>) of Integer;
  function Average (V : Vector_Type) return Float is
  Sum : Integer;
  begin -- Average
    for I in V.Range
      loop
        Sum := Sum + V (I);
      end loop;
    return Float(Sum)/Float(V.Length);
  end Average;
  begin -- Process_Sample_Data
    ...
  end Process_Sample_Data;
```

## INSTRUCTOR NOTES

MODES ARE RESERVED WORDS FOR SPECIFYING PARAMETER MODES.

GO THROUGH EACH MODE, INDICATING THE CALLER/CALLED RELATIONSHIP AND THE KEY POINTS. A MANAGER SHOULD BE ABLE TO READ A PARAMETER LIST FOR A SUBPROGRAM AND UNDERSTAND THE INTERFACE.

NOTE THAT FUNCTIONS CAN ONLY HAVE MODE IN.

THE CALLED SUBPROGRAM CAN, HOWEVER, PASS DATA BACK TO THE CALLER BY ASSIGNING A VALUE TO THE OUT PARAMETER.

# PARAMETER MODES

(Parameter\_List : [mode] Parameter\_Type)

- in (CALLER ----> CALLED)
- ACT AS A READ ONLY VARIABLE
  - IS ASSUMED IF MODE IS NOT STATED

THUS THE CALLED SUBPROGRAM CANNOT RETURN ANY VALUE THROUGH THE in  
PARAMETER TO THE CALLER

- out (CALLER <----- CALLED)
- ACT AS A WRITE ONLY VARIABLE

THUS THE CALLED SUBPROGRAM CANNOT RECEIVE ANY INFORMATION OR DATA  
FROM THE CALLER THROUGH AN out PARAMETER.

- in out (CALLER <-----> CALLED)
- ACT AS A READ/WRITE VARIABLE

THUS THE CALLED SUBPROGRAM BOTH RECEIVES DATA FROM THE CALLER AND PASSES  
DATA BACK TO THE CALLER THROUGH THE in out PARAMETER

## INSTRUCTOR NOTES

REMEMBER, THE SPECIFICATION TELLS YOU WHAT THE SUBPROGRAM DOES, NOT HOW IT DOES IT.

- THE PARAMETERS LISTED IN THE SUBPROGRAM SPECIFICATION ARE CALLED THE FORMAL PARAMETERS. THOSE SPECIFIED IN THE SUBPROGRAM CALL ARE KNOWN AS THE ACTUAL PARAMETERS.
- WHEN THE SUBPROGRAM IS CALLED EACH ACTUAL PARAMETER IS ASSOCIATED WITH A FORMAL PARAMETER. THE TYPE OF THE ACTUAL PARAMETER MUST MATCH THE TYPE OF THE FORMAL PARAMETER. HERE Last\_pt, This\_pt, Time and Velocity ARE LOCAL VARIABLES IN THE CALLING PROGRAM.
- MANAGERS NEED TO BE ABLE TO KNOW
  - THE INTERFACE IS BEING USED CORRECTLY
  - WHETHER ITS A PROCEDURE OR A FUNCTION
  - THE NUMBER OF PARAMETERS
  - THE MODE AND TYPE OF ALL PARAMETERS

# PARAMETER EXAMPLES

## SPECIFICATION (THE INTERFACE):

```
procedure Calculate_Velocity (From, To      : in Point_Type;  
                             In_Time       : in Time_Type;  
                             At_Velocity   : out Float)
```

## FORMAL PARAMETER LIST

## CALL:

```
Calculate_Velocity (Last_Pt, This_Pt, Time, Velocity);
```

FORMALS: From To In\_Time At\_Velocity -- SPECIFIED IN SUBPROGRAM

## SPECIFICATION

ACTUALS: Last\_Pt This\_Pt Time Velocity -- SPECIFIED IN SUBPROGRAM CALL

## INSTRUCTOR NOTES

THE SAME TYPE OF NOTATION USED IN AGGREGATES FOR ARRAYS OR RECORDS IS USED FOR ACTUAL PARAMETER NOTATION.

### POSITIONAL NOTATION:

PARAMETERS MUST BE LISTED IN THE SAME ORDER AS THE FORMAL PARAMETERS

### NAMED NOTATION (THIS IS THE PREFERRED METHOD):

THERE IS AN EXPLICIT ASSOCIATION BETWEEN THE FORMAL AND ACTUAL PARAMETERS, THUS ACTUAL PARAMETERS CAN APPEAR IN A DIFFERENT ORDER THAN THE FORMAL PARAMETERS.

### ADVANTAGES OF NAMED NOTATION

- AVOIDS ERRORS IN PARAMETER POSITION (THE CAUSE OF INSIDIOUS PROGRAM BUGS).

#### EXAMPLE:

MOVE (A, B) -- DOES A MOVE TO B OR B TO A

- IDIOMATIC: IF FORMAL PARAMETERS ARE APPROPRIATELY NAMED THEN THE SUBPROGRAM CALL READS LIKE A SENTENCE.

#### EXAMPLE:

Push ( Element => X, On\_To => Stack);

### DISADVANTAGES OF NAMED NOTATION

- LENGTH: SOMETIMES DETRACTS FROM READING THE PROCEDURE CALL AS A SINGLE ENTITY



# ACTUAL PARAMETER NOTATION

## SPECIFICATION:

```
procedure Calculate_Velocity (From, To : in Point_Type;  
    In_Time : in Time_Type;  
    At_Velocity : out Float)
```

## CALL:

## POSITIONAL

```
Calculate_Velocity (Last_Pt, This_Pt, Time, Velocity); -- ORDER SIGNIFICANT
```

## NAMED

```
Calculate_Velocity (From => Last_Pt, To => This_Pt, In_Time => Time,  
    At_Velocity => Velocity); -- ORDER INSIGNIFICANT
```

INSTRUCTOR NOTES

VG 823.1

6-14i

# OPTIONAL ARGUMENTS AND DEFAULT PARAMETERS

- DEFAULT INITIAL EXPRESSIONS ARE USED WHEN THE MAJORITY OF THE TIME ONE EXPRESSION WILL BE USED FOR AN ACTUAL PARAMETER. FOR EXAMPLE:

```
procedure Set_Display_Size (Rows : in Number_Row_Type := 20;  
                             Cols : in Number_Col_Type := 72) is  
    ...  
begin -- Set_Display_Size  
    ...  
end Set_Display_Size;
```

- WHEN A SUBPROGRAM WITH DEFAULT PARAMETERS IS CALLED, THE ACTUAL PARAMETER(S) CAN BE OPTIONALLY OMITTED

```
Set_Display_Size;  
Set_Display_Size (Cols => 60);  
Set_Display_Size (25);  
-- ROWS = 20, COLS = 72  
-- ROWS = 20, COLS = 60  
-- ROWS = 25, COLS = 72
```

## INSTRUCTOR NOTES

### INFORMATION TO POINT OUT:

1. PROCEDURE VS. FUNCTION
2. FORM OF SUBPROGRAM DETERMINES HOW IT IS CALLED
3. NUMBER OF PARAMETERS, THEIR MODES, AND TYPES (WHICH IS NEEDED TO CALL THE SUBPROGRAM CORRECTLY -- I.E. USE THE INTERFACE CORRECTLY)

AD-A165 314

ADA (TRADEMARK) TRAINING CURRICULUM ADA (REGISTERED  
TRADEMARK) FOR SOFTWARE MANAGERS L201 TEACHER'S GUIDE  
VOLUME 1(U) SOFTECH INC WALTHAM MA 1986

5/5

UNCLASSIFIED

DAB07-83-C-K506

F/G 5/9

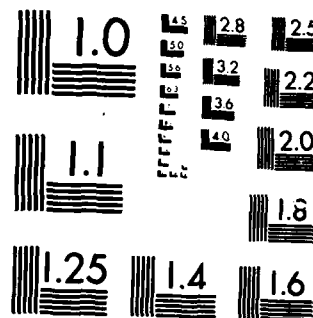
NL

END

FILED

1

DTIC



MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

# CLASS EXERCISE

DETERMINE WHAT, IF ANY, USEFUL INFORMATION CAN BE OBTAINED FROM EACH OF THE FOLLOWING  
SUBPROGRAM DECLARATIONS.

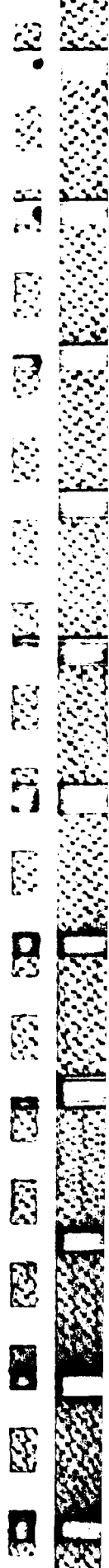
1. procedure Make\_Line\_Available (Physical\_Line : in Physical\_Line\_Type := 1);
2. procedure Rewind\_Tape;
3. function Valid\_Physical\_Line (Logical\_Line : Logical\_Line\_Type;  
Physical\_Line : Physical\_Line\_Type) return Boolean;
4. procedure Calculate\_Median (List : in List\_Type;  
Midpoint : out Scores\_Type);
5. function Hardware\_Clock\_Reading return Hardware\_Clock\_Time;

# INSTRUCTOR NOTES

NOTE THAT Sqrt COULD BE NESTED WITHIN Distance\_Between IF IT IS NOT USED BY ANY OTHER ROUTINES.

VG 823.1

6-161





# NESTING

SUBPROGRAM DECLARATIONS MAY BE NESTED INSIDE ANOTHER SUBPROGRAM.

FOR EXAMPLE:

```
procedure Calculate_Velocity (From, To : in Point_Type;  
    In_Time : in Time_Type;  
    Velocity : out Float) is  
  
    ...  
  
    function Distance_Between (Last_Point, This_Point : Point_Type) return Float is  
        Dx, Dy : Float;  
    begin -- Distance_Between  
        Dx := abs (This_Point(X) - Last_Point(X));  
        Dy := abs (This_Point(Y) - Last_Point(Y));  
        return (Sqrt(Dx**2 + Dy**2));  
    end Distance_Between;  
  
    begin -- Calculate_Velocity  
        Velocity := Distance_Between (From, To)/Float (In_Time);  
    end Calculate_Velocity;
```

NOTE:

- IT IS NOT RECOMMENDED TO WRITE DEEPLY NESTED SUBPROGRAMS.
- AS A RULE, ONLY THOSE SMALL SUBPROGRAMS THAT WILL ONLY BE USED BY THE ENCLOSING SUBPROGRAM SHOULD BE NESTED

## INSTRUCTOR NOTES

IN ADA, ONE GOAL IS TO HAVE ALL INTERFACES EXPLICITLY DEFINED AND TO REDUCE GLOBAL VARIABLES. PASS THEM AS PARAMETERS. ALSO LIMIT THE VARIABLES IN PACKAGES SINCE THESE ARE AVAILABLE TO THE SUBPROGRAMS DECLARED IN THE PACKAGE.

# LOCAL VS. GLOBAL DATA

- VARIABLES DECLARED WITHIN A SUBPROGRAM ARE KNOWN AS LOCAL VARIABLES:

procedure P is

Local : Integer; -- local variable

begin -- P

...

end P;

Local only known here

- VARIABLES KNOWN WITHIN A SUBPROGRAM BUT DEFINED BY THE CALLER (INSTEAD OF LOCALLY) ARE KNOWN AS GLOBAL VARIABLES
  - ASSIGNMENT TO GLOBAL VARIABLES (SIDE EFFECTS) IS ALLOWED WITHIN BOTH FUNCTIONS AND PROCEDURES
  - SUCH PRACTICES ARE THE MAINTENANCE PROGRAMMER'S NIGHTMARE
  - VALUES YOU WISH TO CHANGE SHOULD BE PASSED AS in out PARAMETERS

## INSTRUCTOR NOTES

### POINT OUT:

1. IT'S A PROCEDURE WITH NO PARAMETERS
2. THE DECLARATIVE, EXECUTABLE AREAS
3. A FUNCTION CALL TO Distance\_Between WHICH IS IN THE WITH'ED PACKAGE
4. AFTER EXECUTION IS FINISHED THE LOCAL VARIABLES OR CONSTANTS FOR Compute\_Tracking\_Data CEASE TO EXIST. THE SAME THING HAPPENS FOR THE SUBPROGRAM Distance\_Between. EVERY TIME YOU CALL IT, YOU START ANEW.

## SUBPROGRAMS AS MAIN PROGRAM UNITS

```
with Text_IO;
with Vector_Services; use Vector_Services;

procedure Compute_Tracking_Data is

    Distance           : Float_Type;
    Last_Point, Current_Point : Point_Type;

    package Flt_IO is new Text_IO.Float_IO (Float);
    procedure Get_Point (P : out Point_Type) is separate;

begin -- Compute_Tracking_Data

    Text_IO.Put ("Enter coordinates of last point : ");
    Get_Point (Last_Point);
    Text_IO.Put ("Enter current coordinates : ");
    Get_Point (Current_Point);

    Distance := Distance_Between (Last_Point, Current_Point);

    Text_IO.Put ("Distance between points is ");
    Flt_IO.Put (Distance);

end Compute_Tracking_Data;
```

INSTRUCTOR NOTES

ALGORITHM IS A BASIC SORT USING A SIMPLE INTERCHANGE.

## SUBPROGRAMS AS DEFINITIONS OF ALGORITHMS

```
procedure Sort (List : in out List_Type) is
    Temp      : Scores_Type;
    Sorted    : Boolean := False;
begin -- Sort
    while not Sorted loop
        Sorted := True;
        for I in List'first .. List'Last - 1 loop
            if List(I) < List(I + 1) then
                Temp := List (I);
                List(I) := List (I + 1);
                List(I + 1) := Temp;
                Sorted := False;
            end if;
        end loop; -- for
    end loop; -- while
end Sort;
```

INSTRUCTOR NOTES

ABSTRACT DATA TYPE WITH ITS CORRESPONDING OPERATIONS.

THE THREE PROCEDURES ARE THE ALLOWABLE OPERATIONS ON THE TYPE Message\_Type.



## SUBPROGRAM AS OPERATOR ON A TYPE

```
package Message_Switch is
  type Message_Type is private;
  procedure Get_Message (Message : out Message_Type);
  procedure Process_Message (Message : in out Message_Type);
  procedure Send_Message (Message : in Message_Type);
private
  ...
end Message_Switch;
```

INSTRUCTOR NOTES

REQUIREMENTS NEED TO BE STATED WELL UP FRONT. ADDITIONAL TIME SPENT IN REQUIREMENTS AND DESIGN IS COMPENSATED BY EASIER CODING AND DEBUGGING (SINCE THE PROBLEM IS BETTER UNDERSTOOD).

# SOME POTENTIAL PITFALLS - SUBPROGRAMS

- USING SUBPROGRAMS TO DEFER DESIGN DECISIONS (I.E., PROCESSING "TBD"). FOR

## EXAMPLE:

IN A MESSAGE SWITCH, WE NEED A Log\_To\_Database ACTIVITY. FOR THE PACKAGE SPEC WE HASTILY DEFINE AN INTERFACE, AND MOVE ON. BY THE TIME WE GET AROUND TO IMPLEMENTING Log\_To\_Database, WE DISCOVER THAT IT NEEDS TO KNOW WHICH BACK-UP DEVICE IS CURRENTLY ON-LINE. TRADITIONALLY, THAT'S WHY PATCHES ARE INTRODUCED AND WHY EVERY VARIABLE IS MADE PUBLIC -- JUST IN

## CASE:

HINT: AT THE TIME AT WHICH AN INTERFACE IS DEFINED, ONLY THE ALGORITHM SHOULD BE "TBD." THE FUNCTION SHOULD BE WELL-SPECIFIED.

INSTRUCTOR NOTES

VG 823.1

6-22i

# SUMMARY

## SUBPROGRAMS

- BASIC EXECUTABLE ADA PROGRAM UNITS
- TWO FORMS: PROCEDURES AND FUNCTIONS
- COMPOSED OF A SPECIFICATION AND A BODY
- THE SPECIFICATION CAN OCCUR IN A PACKAGE SPECIFICATION
- MAY BE NESTED IN ANOTHER SUBPROGRAM
- ARE NOT FOR DEFERRING DESIGN DECISIONS

## INSTRUCTOR NOTES

THIS EXERCISE IS INTENDED TO PULL TOGETHER A LOT OF THE ADA DETAILS PRESENTED SO FAR AND TO SET A FOUNDATION FOR THE LARGE EXERCISE IN THE NEXT MAJOR SECTION.

METHOD OF PRESENTATION (QUESTIONS TO ELICIT FROM CLASS):

1. WHAT PROGRAM UNIT IS SHOWN (SUBPROGRAM -- PROCEDURE)
2. WHERE IS THE DECLARATIVE PART AND WHAT SHOULD BE IN IT (SHOULD HAVE DECLARATIONS -- LOCAL)
3. WHAT IS THE DECLARATION (AN OBJECT CALLED C OF TYPE CHARACTER)
4. WHERE IS THE EXECUTABLE PUT AND WHAT SHOULD BE IN IT
5. IDENTIFY THE KINDS OF STATEMENTS (INFINITE LOOP, IF)
6. WHAT DOES THIS PROCEDURE DO (IT GETS A COMMAND ABBREVIATION, EXPANDS AND PRINTS THE CORRESPONDING COMMAND)

# CLASS EXERCISE

```
with Text_IO; use Text_IO;
procedure Expand_Command is
  C : Character;
begin -- Expand_Command
  loop
    Get (C);
    if C = 'E' or C = 'e' then
      Put ("Edit");
    elsif C = 'L' or C = 'l' then
      Put ("List");
    elsif C = 'H' or C = 'h' then
      Put ("Help");
    else
      Put ("No Command");
    end if;
  end loop;
end Expand_Command;
```

# INSTRUCTOR NOTES

THIS SLIDE IS THE SAME AS THE PREVIOUS SLIDE EXCEPT THE IF HAS BEEN REPLACED BY A CASE STATEMENT. ASK THE STUDENTS TO ASSESS WHICH IS EASIER TO UNDERSTAND (AND SIMPLER).

NOW ASK THE CLASS WHAT WOULD HAVE TO BE DONE IF WE WANTED TO ADD A NEW COMMAND ABBREVIATION FOR EXPANSION. ANSWER: ADD ANOTHER CHOICE OPTION IN THE CASE STATEMENT. NO CHANGE TO THE LOCAL VARIABLE.

POINT OUT THIS CHANGE AFFECTS THE EXECUTABLE PORTION OF THE ALGORITHM. ASK THE CLASS IF THAT'S REALLY SOMETHING WE WANT TO DO. DO WE RUN THE RISK OF INTRODUCING ERRORS AS A RESULT OF ONE CHANGE? (ESPECIALLY IF THIS WAS A CRITICAL OR INTRICATE ALGORITHM).



## CLASS EXERCISE (Continued)

```
with Text_IO; use Text_IO;

procedure Expand_Command is
  C : Character;
begin -- Expand_Command
  loop
    Get (C);
    case C is
      when 'E' | 'e' => Put ("Edit");
      when 'L' | 'l' => Put ("List");
      when 'H' | 'h' => Put ("Help");
      when others  => Put ("No Command");
    end case;
  end loop;
end Expand_Command;
```

**END**

**FILMED**

4-86

**DTIC**